

THE ESSENTIAL GUIDE TO

# Flash游戏编程指南

Building Interactive Entertainment  
with ActionScript 3.0

- ▶ BUILD A FULL-FEATURED, MODULAR ACTIONSCRIPT 3.0 GAME FRAMEWORK THAT YOU CAN USE TO CREATE YOUR OWN GAMES
- ▶ TEN FULL GAME PROJECTS ARE DISCUSSED IN DETAIL WITHIN THE BOOK, INCLUDING, SPACE SHOOTERS, DRIVING GAMES, AND PUZZLE GAMES
- ▶ LEARN ADVANCED GAME DEVELOPMENT TECHNIQUES INCLUDING, BITMAP RENDERING, PIXEL-LEVEL COLLISION DETECTION, A.I., BLITTING, SCROLLING, PARTICLES, REUSABLE FRAMEWORKS AND MUCH MORE

JEFF FULTON AND STEVE FULTON





9Ria.com 全球最大的 RIA 中文开发者社区

# Flash 游戏编程指南

创建 ActionScript 3.0 互动娱乐

Jeff Fulton

Steve Fulton

译林军书组敬献

翻 译 : aserrewin, kenjor, yyluo-阿树, sun11086-0025, peichao01, cosmos53076, yangjh415, Pizzaman, 享受生活, 心月不眠, Mr.Star

经过译林军书组的兄弟数个月的努力, 我们终于能把这本书呈现到各位兄弟面前。

译稿将以连载的形式在接下来的一个多月内持续发布, 每日更新。

诚然, 由于书组的兄弟时间, 精力以及水平有限, 错漏难免。希望各位兄弟能将校对意见跟帖提出。

或加入本书交流 **Q 群: 32831062** 一起讨论

论 坛 交 流 : <http://bbs.9ria.com/thread-68391-1-1.html>



译林军书组敬献《Flash 游戏编程指南》



## 目录

### 第一部分：基本游戏框架

第一章： “第二游戏”说	5
第二章： 创建一个 As3 游戏框架	56
第三章： 创建超级点击	150

### 第二部分：游戏实践

第四章： 御空加农炮的基础架构	189
第五章： 构建御空加农炮游戏循环	226
第六章： 预备！坦克大战！	265
第七章： 构建坦克大战游戏	310
第八章： 休闲智力游戏--魔法色块	411
第九章： 骰子游戏王	455
第十章： 滚屏游戏世界	502
第十一章： 制作绝佳的反应力游戏	568
第十二章： 制作一个 Viral Game： 隧道惊魂	663

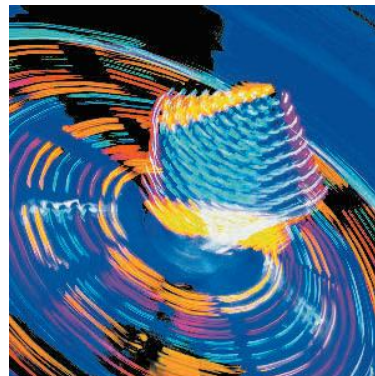


## 第一部分

# 基本游戏框架

---

在第一部分中，你将学习到如何建立一个优秀的游戏框架。



## 第一章

# 第二游戏说

嗯，第二个游戏？那第一个游戏哪去了？当然，你必须制作你的第一个游戏，但是不可避免的，也许你的第一个游戏并不是你所期望的那个样子，但它还是成了这样糟糕的东西。先别急着自责，也许你会抽时间来修改，或者浪费时间在沮丧上，甚至因此都不能完成它。但是，这恰恰是我们需要你做到的——做完它，然后开始做第二个。这是你将来能够制作优良游戏的唯一途径，这就是所谓的“第二游戏”说。

这本书的初衷是我们相信能够帮助你完成你的游戏并开始着手于下一个。这对于我们来说也是很重要的，因为无论是开发者还是 Flash、Flex 讲师又或者是游戏的设计者和程序员，实际上都要好好完成他们的每一个项目。我们都喜爱玩在线的 Flash 游戏。每次你做完一个游戏，不管是不是真的像你的想的那么好，把它弄出来，不管怎么说，你都从中受益了，你会变得更优秀，我们也能有更优秀的游戏可以玩。Flash 游戏界的技术水平也能提高一点点。所以你的任务就是做完它，从中吸取教训，学习经验。然后开始进行下一个工程。做完第二个，就做第三个，做完第三个就做第四个、第五个，直到 Flash 游戏成为你生命的一部分，到那时整个互联网都会因为你的下一个作品闹的沸沸扬扬。

然而，你的第一个游戏不是指那种只是一个雏形，一个测试版，或者一个半成品的玩意。它应该包含完整的结构。如果不是的话，你永远都要在“第一个游戏”面前磨蹭，那会导致一种称为“开发的地狱”的情况：你的余生永远被一个不完整的游戏折磨。（囧）

## 制作游戏是一个反复的过程

制作游戏与开发其他软件的一个主要不同就是：好灵感不都等同于成功的作品。换句话说，有些特别创意的最初想法在游戏设计上根本一无是处，没人会玩。



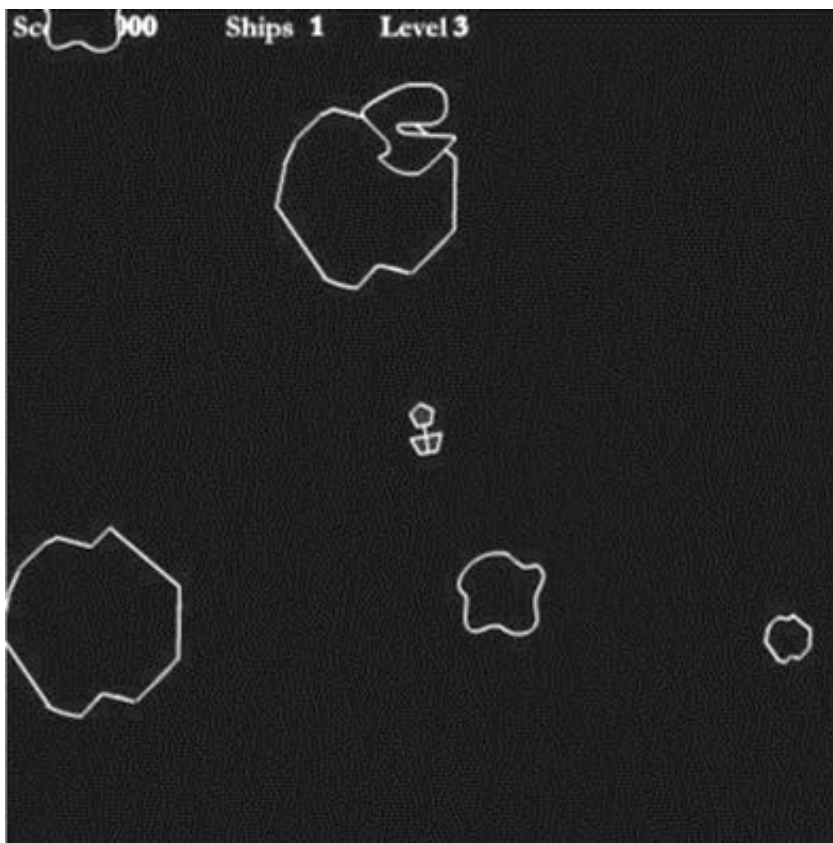
这么说吧，你想做一个CD烧录软件，一直到制作完成你都可以很好的贯彻你的目的。一旦你可以把文件拷进CD里，并且让CD设备能够读取它。你的工作实际上差不多就算完成了，也许可以再细化一些外观什么的。无论怎么样，你的目的和你的作品一致了。

这种清晰的流程不是每次制作游戏都能碰到的。而且太简单或者太复杂的游戏并不是任何人都想玩。比如说，仅仅只是想做一个能让人玩的游戏是不够的。别着急，第一次都这样。

需要琢磨的地方来了，这时要反复的进行思考。有时候小小的改变或者升级甚至只是在开发流程上要一个小聪明就能让你的游戏程序变得有趣起来。而且，这个反复的思考的作用你要有更高的重视。除了制作一个创意独特的游戏，还要考虑到你的技术水平。总之，量力而行。每当你完成了一个游戏的制作，你都要改良你游戏的代码、流程、模式、数据结构、创意、设计等等。这就是为什么我跟你啰嗦制作第二个游戏很重要。一旦能够写第二个游戏的时候就说明你开始自我提高了，而且还会随着下一个游戏的制作再次提高。

下面的图片1-1和1-2是两个游戏的图示。Zeno Fighter (图示1-1) 是第一个移动射击风格的游戏，Retro Blaster (图示1-2) 是从上一个游戏扩展而来的。所以说，如果没有做完Zeno Fighter的话，也就不会有Retro Blaster了。Zeno Fighter不是一个雏形，半成品或者测试版本：它是一个完整的游戏。之后所有的游戏开发都是利用它创建的。当它完成时，它使你学到的编程基础和经验足够你弄出第二个游戏。

好了，言归正传，第二个游戏不一定要和第一个游戏是同一流派的。实际上，在本书中，我们会一起来制作很多不同类型的游戏，为的就是靠这个反复制作的过程来提高。





图片 1-1. A Zeno Fighter screen shot



图片 1-2 Retro Blaster

## 要明白你为什么想做游戏

现在，给你几分钟的时间来想一个问题：为什么你想要做游戏？

既然你已经买了这本关于制作 Flash 游戏的书，所以你可能早就想过这个问题了。就算没有，现在想也来得及。知道做一件事情的动机是很重要的。如果你只是冒冒失失的走，那么走起来可就没完没了了。必须给自己一个目标和你确立这个目标的合理动机。

- 你想因此来学习一些高级的 ActionScript3.0 语言？
- 还是仅仅为了工作需要？
- 又或者这是学校老师布置的任务？
- 甚至你仅仅是想在你朋友面前显摆显摆？
- 还是说，你是一个经验丰富的程序员或游戏程序员。想学习如何用 ActionScript3.0 来制作游戏？

了解为什么要做游戏是很重要的，因为这是达成目的的必要条件，合理的目标与动机。一个空想型的游戏开发者可以花好几个小时读一本书的一个章节而只是打那么几行的代码，并且还自我感觉非常不错。对他们来说，克服空想的毛病并投入大量时间来制作游戏是很痛苦的。我们就是来让你尽可能舒服地制作你的第一个游戏，然后是第二个、第三个。



只要一会功夫儿就可以设计一个游戏并实现它。实际上，你可能觉得还没有一个称手的制作工具或者是一个好的执行流程又或者是，你还没有粉丝，这些都不重要，仅仅靠一些相当简单的途径就可以反馈你的工作成果了，所以不要找借口让一个半成品游戏放在你的硬盘上，你没有理由停滞不前。

## 这本书是给谁写的？任何需要它的人！

没错，就是这样，任何想制作游戏的人都可以。实际上，对于已经有一些面向对象语言经验的人比如 ActionScript2 或 3.0(AS2 或 AS3),C++,C#,JAVA,或者是 JavaScript 的程序员。我们仅仅是影响制作一些游戏的方法罢了，不会花大量的时间去教你程序的基础知识。你的直觉应该可以帮你从后面的程序代码或描述中挑选主要的概念，但是我们不能在其他语言上弄的太久，在这本书里，我们要做 Flash 游戏，就需要写 AS3 代码。

## 那么我们现在干嘛不开始呢？

当然，我们会开始的。我们开始写 AS3 的游戏程序。还有很多其他的资源可以教你 As3 语言的基础知识，比如《Foundation ActionScript 3.0 for Flash and Flex》(ISBN:978-1-4302-1918-7)，还有《Foundation Game Design with Flash》(ISBN:978-1-4302-1812-0)。在这本书里，我们要教你一些基本概念，在这之后你可以用这些代码来制作、修改你的游戏。闲话少说，我们将随着学习的进行来了解一些基本的 AS3 概念。当然了，有能力的读者应该可以很容易的掌握这些概念，这样的话你可以很快的学习它们了。

## 比较 AS3 和 AS2

“AS3”是 ActionScript3.0 的简称。并且是 Flash 和 Flex 游戏与应用程序的核心语言。它基于 AS2 并重新改写了 ActionScript2，所以如果你只会 AS2，你也可以使你很快的掌握 AS3 的基础并开始学习这本书。

早在 90 年代 Flash 4 发布的时候 Flash 就开始支持一些简单的脚本了。Macromedia 在 Flash 5 中引进了 ActionScript，并在经过了几个小波折之后最终成为 Flash 8 版本的 ActionScript 2。ActionScript 2 很不错，你可以用面向对象的方法开发 Flash，使用事件机制，支持位图等等。然而，它被一次又一次的被重写，支持的函数越来越多，就像你想的那样，它不断被完善。当它对应用的支持越好的时候，它在浏览器上运行的性能也越来越差。

正是由于这一点，Flash 的开发团队回到了起点，重新设计 ActionScript 语言，他们努力的试图修复 AS2 的种种问题(和它庞大的体积)。而 AS3 的开发完成是对 AS2 的一个重大意义的飞跃。

你不能死抱着 AS2 不放因为它过时了

不能逃避 AS3 比 AS2 更完善的事实。许多在线的教程仍然坚持使用 AS2 代码，你会发现许多游戏的函数和结构都是用 AS2 写的，还有许多开发者仍然非常依赖 AS2。但是，为什么要改变这一点呢？主要的问题就是 AS2 它过时了，Adobe 已经不想再支持它了，因为这会使 Flash 开发者转移到 AS3 上更困难。事实上，对于游戏开发者来说，AS3 有许多飞跃性的改进。相比 AS2 来说，有以下几点：

- **速度**：AS3 的编译执行的比 AS2 简单多了，一个新建的 Flash 文件从创建到执行编译，它都会被最即时的完全执行。许多游戏的运行性能快就是因为这个小因素。AS3 的执行速度是 AS2 的 2



到 5 倍(不相信可以自己测试一下它们的性能差距)。

- **容错处理/调试**：在 AS2 中，错误的处理和调试都是通过很多 trace 语句来验证。AS3 新增了一个强大的编译器来寻找逻辑、语法以及运行时间错误的系统，来帮助诊断那些看起来很好的代码究竟哪出了问题。
- **高级的事件模型**：显然很多游戏并不是简单的傻等着用户来输入操作，而是创建一个充满了触发事件的世界，这需要在程序中加入许多的事件句柄。这样的话，一个可靠的事件模型就是必要的了。而 AS2 使用的事件模型不是很成熟。在 AS3 中，事件被建立在核心中，使用起来十分的方便，并且这样可以让开发者开发出更容易设计更有逻辑性的软件。这也便于维护。

还有许多其他的区别，但是这些已经足够让你决定用 AS3 来开发游戏了。

## 这本书你需要用到什么

FlashCS3 或者是 Flex or Flash Builder。Flex SDK 或者任意一个程序编辑器 ( i.e.,Flash Develop )。总之，你自己看哪个更方便使用哪个。

我们会尽量描述这本书里的代码在多种开发工具上是如何运行的，但是需要注意几点。你可能要对书中的代码作出一点点的修改使其适合你的开发工具。

## 制作 AS3 游戏的框架

在这本书里，我们会建立一个游戏框架并进行一些使用设置。然而，这个框架不是游戏引擎。一个游戏引擎是通过一系列封装方法来制作游戏的，比起软件工程来它更像是装饰学。你只是使用了引擎提供的方法，而不是你自己写的代码。

游戏框架的一个好处就是能帮你组织代码段和游戏函数，你可以相当容易的写出易于扩展的游戏代码。然而，框架的基本代码是不变的。即使你可以使用一个开源框架的引擎，但是它的框架运作方式你怎么可能都了解呢。但是我们自己写的框架就好多了，它是专门用来使你自己的游戏运作的更有效而编写的，所以它和谐多了。

当我们要开始编写自己的游戏框架的时候，你应该专心致志的来跟着我们学习。当你变得擅长制作游戏后，你就可以自由制作你需要的框架了(甚至是一些临时拼凑的框架)。本书的目的不是宣传我们的框架，而是通过这种方法来让你理解如何创建一个你自己的游戏框架。同时也让你明白一个可用性好的框架可以帮你制作更多的游戏。

## 面向对象方法

许多的近现代程序语言都是面向对象的，既然游戏框架也是基于这种概念，我们就来简短的解释一下它们并说明为什么面向对象是必要的。这种概念可以让我们有计划地开发游戏，而且也是足够的灵活以便应付这样或那样你当初没有想到的问题。



**面向对象(OO)方法**是一种程序的开发思想以使开发者能够通过清晰的定义，使其程序和组件逻辑更紧密。有很多形式的 OO 设计。但是我们最多使用就是封装与继承。

## 封装

**封装**是一种能把你的程序分解成组件的方法，组件可以包括隐藏或者显性（私有或公有）的属性和方法。这种模式广泛的包涵在了我们的游戏框架中。许多游戏的框架服务（示例：声音控制）都需要保持更新。但是由于他们使用了通用接口，新的版本就会完整的与旧版本结合。封装使得这成为可能。

我们会使用包（package）的概念来学习封装。包是一个将一些类聚集在一起的实体，它们有相似的目的。游戏框架的本身就是一个包，每一个游戏都有它自己的包。包中的每一个类都与包中其他的类密切相关。所以使用之前，确保它们被明确的导入。

## 继承

**继承**的概念就是从第一个对象（基类）中通过继承的方法来创建一个更专门的子集的方法，（译者注：类似于论坛的子版块）只需要定义一个新的子类并修改其属性与方法就可以了。例如，创建一个基本的屏显类来应用于大多数情况（示例：标题显示或者是介绍文本的显示）。然而，也许你突然需要一个只是有些不同方法和属性的屏显类。这时我们就可以用继承来实现。继承可以使组件重用和游戏开发更简易。

你会看到通过使用继承我们显著的减少了在新建游戏项目时粘贴复制代码的次数，我们发现（在我们的项目里）使用继承减少了我们开发新项目所用时间的百分之 10 到百分之 30。那说明什么？说明继承是个好主意，用得好就能帮你省钱。

## 使用面向对象方法设计游戏

使用面向对象的游戏开发设计包括把你的游戏项目分解成一个个功能组件，并给这些组件定义接口。现在坐下来花一点时间来考虑你需要的什么样的功能。

让我们假设要制作一个太空射击类的小游戏，对于这种类型的游戏，你可能要考虑的部分有以下几点：

- Player ship -玩家的飞船
- Player shots -玩家的子弹
- Asteroids (3 sizes) -小行星（2 种）
- UFO (2 sizes)- UFO（2 种）
- UFO shots -UFO 子弹
- Scoreboard-计分板

一些基本的思考，你就建立了你的组件模型。它还不是代码，但是你可以用这些东西很快的组织你的想法。然而，我希望你能在写任何代码之前能多想想组件在面向对象的逻辑中应该包括的属性（变量）和方法（函数）。你可以设计的再长远一些。

玩家的飞船可能包括



- Properties\_属性
  - Speed\_速度
  - Angle\_角度
  - Rotation\_转向
- Functions\_函数
  - turn()\_转向函数
  - thrust()\_冲刺函数
  - fire()\_射击函数

玩家的子弹可能包括

- Properties\_属性
  - Speed\_速度
  - Angle\_角度
  - Life\_生命
- Functions\_函数
  - move()\_移动函数

小行星可能包括

- Properties\_属性
  - Size\_大小
  - Speed\_速度
  - Angle\_角度
  - Rotation\_转向
- Functions\_函数
  - move()\_移动函数
  - explode()\_爆炸函数

UFO 们可能包括

- Properties\_属性
  - Size\_大小
  - Speed\_速度



- Angle\_角度
- Firing rate \_射击频率
- Functions\_函数
  - shoot()\_射击函数
  - move()\_移动函数

UFO 子弹可能包括

- Properties\_属性
  - Speed\_速度
  - Angle\_角度
  - Life \_生命
- Functions\_函数
  - move()\_移动函数

计分板可能包括

- Properties\_属性
  - Score\_分数
  - Ships left \_剩余生命数
  - Level\_等级
- Functions\_函数
  - updateScore()\_更新分数
  - updateShips()\_更新生命数
  - updateLevel()\_更新等级

你可以看到这种形式的计划列表可以阐明游戏运作时的大部分内容。更进一步地，你还要用这种设计模式来创建一个管用的组件。现在看起来这个工作要完成了，那我们就开始创建游戏吧。你会看到这种计划模式是如何高效帮助你用 AS3 来建立游戏项目的，你也会注意到这非常的灵活。当一个游戏需要高性能或者一个合适的文件大小的时候，这种计划模式就会起效果。我们也会尽可能地不背离这种原则，当然，如果我们不用的时候，我会给你一个好理由。



## 创建基本游戏框架

现在，我们要建立一个非常基本的游戏框架版本来做例子，用以说明本书中其他的游戏是如何被创建的。这个框架会被更新，修改，改良或者增加更复杂的东西。但是我们需要在开始地方，就是这个基本的游戏框架。我们会用代码建立一个非常简单的游戏，这个游戏本身不是很重要，重要的是它的代码结构。别担心做错了，我们会帮你随着我们的脚步进行修正。现在要说明的是这个简单游戏的基本原理。

这个游戏不是一个简单的点击按钮触发事件的测试代码。而是一个用基本的游戏框架搭建起来的完整游戏。这个游戏被用于举例说明一个框架的大部分主要结构。首先就是**状态循环**。我们的游戏大概有三个状态，一个是初始状态，一个是游戏状态，还有一个是游戏结束的状态。根据游戏运行的需要来确定它的当前运行状态，这个就是状态循环。比如，在游戏状态，需要等待玩家点击按钮，当按钮被按下的时候，游戏状态转变为结束状态。这个状态循环不断地检测现在游戏到底进行到哪一环节了，根据环节的不同来采取恰当的行为。而这个反复检测状态检测的东西被称作**游戏频率**，是游戏框架的第二个核心部分。还有一点，当按钮被按下的时候，这个处理鼠标点击函数的东西被称作**事件模型**。这个是游戏框架的最后一个核心部分。让我们更详细地来解释一下这三个部分。

### 状态循环

状态循环是游戏行为控制的交警。一个非常基本的使用方式就是使用常量或 Switch 方法。常量是不会改变的。它们的声明就像变量一样，但是不是用 var 关键字。后面的代码会演示这种方式。而 var 关键字是给变量用的，变量就是在程序执行是可以改变的数值。

下面的代码中建立了几个类中用到的状态，这些状态会控制游戏的流程。我们用 const 关键字来声明它们，而且他们是一种静态常量，并且可以在不用建立实例的情况下被其他类调用。

```
1. public static const STATE_INIT:int = 10;
2. public static const STATE_PLAY:int = 20;
3. public static const STATE_GAME_OVER:int = 30;
4. public var gameState:int = 0;
```

前三个状态表示了我们游戏运行时可能出现的状态。我们把它们设置为常量。因为我们的游戏会依靠这些不会被修改的常量来控制游戏运行的状态。顺便说一下，这个示例非常简单。你可以为你的游戏设置更多的状态，实际上，还有一种多状态嵌入到一种状态的情况。（第八章和第九章会说明这种状态的嵌套）。最后的那个语句。public var gameState:int = 0;建立了一个变量来保存当前游戏保持的状态。下面的函数 gameLoop(),是游戏的主要循环，它表示了游戏的频率（后面会说明）。通过 Switch 方法来决定当前游戏应该保持哪一种状态（通过检测 gameState 变量）。

```
1. public function gameLoop(e:Event):void {
2.     switch(gameState) {
3.         case STATE_INIT :
4.             initGame();
5.             break
```



```
6.         case STATE_PLAY:
7.             playGame();
8.             break;
9.         case STATE_GAME_OVER:
10.            gameOver();
11.            break;
12.    }
13.}
```

这个状态循环的建立是有充分的理由的,如果没有它,你可能不得不用大量的 Boolean( true 或 false ) 变量来了解程序流。也许那可以用于一些十分简单的游戏。但是更多的情况下,使用过多的 Boolean 变量是一场噩梦。假设一个 Boolean 变量没有弄对,整个系统就玩完了。状态机制通过使游戏始终保持单态(也可能是嵌套状态)来解决这个问题。

## 游戏频率

游戏频率也可以啰嗦成通过每隔一段时间来检查游戏状态的机制。注意前面的代码有一个函数叫 gameLoop() 的。从名字上看来,估计你能猜出它是游戏执行的一种循环逻辑。这个游戏的频率通过一个 ENTER\_FRAME 事件来反复的调用 gameLoop ( ) 函数。代码看起来就像这样:

```
1. public function Game() {
2.     addEventListener(Event.ENTER_FRAME, gameLoop);
3.     gameState = STATE_INIT;
4. }
```

这种通过 EnterFrame 事件来处理游戏循环的方法在 Flash 中很常见。这是本书中介绍的最基本的建立游戏频率的方法。它是一个正确的方法,但是还有很多其他有效的方法。然而,这个简单的方式很好的说明了游戏频率是怎么运作的。游戏频率通过一种很规律的方式反复调用游戏循环。我们已经把这个游戏的状态设置为 STATE\_INIT 了,然后我们的 gameLoop ( ) 函数就会在下次执行 initGame ( ) 的方法了。

## 事件模型

事件就是用来告诉游戏:有一个有趣的事发生了,你该有所行动了。AS3 定制了很多事件。我们要监听(或者更复杂的规则)一个鼠标点击按钮的事件。虽然这个事件看起来不怎么有意思。但是这个基本的示例可以让我们重用和改良其他我们需要的事件。无论是 AS3 内置的还是我们自定义的。现在来看看建立一个事件监听的代码。initGame() 函数定义了这个游戏的事件模型。下面的一行代码告诉我们的类要监听这个鼠标点击事件。

```
1. stage.addEventListener(MouseEvent.CLICK, onMouseClickEvent);
```

第一个参数 ( MouseEvent.CLICK ) 是事件的名称。然后第二个参数 ( onMouseClickEvent ) 是事件触发是要调用的函数名称。我们把记录点击次数的变量设为 0 并把游戏状态设为 STATE\_PLAY, 这样一来 gameLoop ( ) 函数就知道下一步该怎么做了。

```
1. public function initGame():void {
```



```
2. stage.addEventListener(MouseEvent.CLICK, onMouseClickEvent);
3. clicks = 0;
4. gameState = STATE_PLAY;
5. }
```

如果游戏状态等于 STATE\_PLAY 的话，playGame ( ) 函数就会被 gameLoop ( ) 函数调用。这个函数检测点击的次数是不是大于 10 次。如果是的话就把游戏状态设置为 STATE\_GAME\_OVER。

```
1. public function playGame() {
2.     if (clicks >=10) {
3.         gameState = STATE_GAME_OVER;
4.     }
5. }
```

下面的代码是每次触发鼠标点击事件时调用的函数。它随着每一次调用把记录鼠标点击次数的 clicks 变量的值加 1。

```
1. public function onMouseClickEvent(e:MouseEvent):void {
2.     clicks++;
3.     trace("mouse click number:" + clicks);
4. }
```

最后，endgame ( ) 函数是在游戏状态被 playGame ( ) 函数改为 STATE\_GAME\_OVER 的时候被 gameLoop ( ) 函数调用的。这里要做的就是对 MouseEvent.Click 事件采取 removeEventListener() 方法 ( 移除监听 )。为什么要这样做呢？因为在游戏结束的时候所有的事件监听都应该被清除。当我们把游戏状态设置为 STATE\_INIT 时游戏就会自动的再次开始。然后我们输出一下这个游戏的结果来告诉玩家发生什么事了。

```
1. public function gameOver():void {
2.     stage.removeEventListener(MouseEvent.CLICK, onMouseClickEvent);
3.     gameState = STATE_INIT;
4.     trace("game over");
5. }
```

现在，我们可以把这些代码组合起来变为一个简易的游戏了。我们所写的代码都要收集到一个名为 Game 的类中。这个代码可以被 Flex 使用或者直接在 Flash IDE 中作为一个 AS3 应用的类文档 ( 主类会在程序运行的时候被执行 )。

下面是程序流程的简单描述：

1. Game 类被实例化。
2. 构造函数被调用。
3. 构造函数设置每一帧的游戏频率。



4. 构造函数设置游戏状态为 STATE\_INIT。
  5. 游戏频率调用 gameLoop ( ) 函数。
  6. gameLoop ( ) 函数根据游戏状态的变量来选择函数。
  7. initGame()函数在被调用的时候，鼠标事件的监听被建立，变量进行重置。
  8. initGame ( ) 函数把游戏状态设置为 STATE\_PLAY。
  9. 游戏频率调用 gameLoop ( ) 函数。
  10. gameLoop ( ) 函数根据游戏状态的变量来选择函数。
  11. playGame ( ) 函数在没有达成游戏结束条件的时候被调用（结束条件：clicks>=10）。
- 这一步在没有达成条件之前会一直执行。除非 MouseEvent.click 事件使得条件成立了。
12. 如果 clicks 大于 10 了，playGame ( ) 函数就会把游戏状态改为 STATE\_GAME\_OVER。
  13. gameOver ( ) 函数被调用，清空监听，输出结果并重新开始游戏。

（译者注：这里有点问题，原书上写了 16 步，但是 13 到 15 跟 4 到 6 步是一样的，而且也不通顺，所以我去掉了，如果看过的人知道是不是刊印错误请告诉我）

总之，当游戏运行的时候，鼠标左键点击按钮只是简单的增加点击次数，而 playGame ( ) 函数的作用就是查看点击次数并在超过 10 次的时候更改游戏状态。

不管你相信不相信，现在我们就算是把这个游戏弄完了，这可能是最简单的游戏了。所有的玩家只要点击鼠标 10 次就能赢！

这里有一些 AS3 代码中需要注意地方：

- Package { }：不像 AS2 的类，所有 AS3 的类需要在包中定义，如果你没有给包指定一个名称，AS3 会使用默认的包名，我们一会要创建一个包，现在往下看。
- import：在 AS3 中，你必须导入所有不在你包中定义的类。跟 As2 不同的是有一些类是默认被导入。
- Public class Game extends flash.display.MovieClip{}：一个文档类需要完全的继承影片剪辑类。与 AS2 不同的地方就是你可以只继承 MovieClip 类。

那就是所有你需要知道的拉！现在第一个游戏的所有代码就完成了：

```
1. package {  
2.     import flash.display.MovieClip;  
3.     import flash.events.Event;  
4.     import flash.events.MouseEvent;
```



```
5.    import flash.display.*;
6.    import flash.events.*;
7.    import flash.net.*;
8.    public class Game extends MovieClip{
9.        public static const STATE_INIT:int = 10;
10.       public static const STATE_PLAY:int = 20;
11.       public static const STATE_GAME_OVER:int = 30;
12.       public var gameState:int = 0;
13.       public var clicks:int = 0;
14.       public function Game():void {
15.           addEventListener(Event.ENTER_FRAME, gameLoop);
16.           gameState = STATE_INIT;
17.       }
18.       public function gameLoop(e:Event):void {
19.           switch(gameState) {
20.               case STATE_INIT :
21.                   initGame();
22.                   break
23.               case STATE_PLAY:
24.                   playGame();
25.                   break
26.               case STATE_GAME_OVER:
27.                   gameOver();
28.                   break;
29.           }
30.       }
31.       public function initGame():void {
32.           stage.addEventListener(MouseEvent.CLICK, onMouseClickEvent);
```



```
33.         clicks = 0;
34.         gameState = STATE_PLAY;
35.     }
36.     public function playGame() {
37.         if (clicks >=10) {
38.             gameState = STATE_GAME_OVER;
39.         }
40.     }
41.     public function onMouseClickEvent(e:MouseEvent) {
42.         clicks++;
43.         trace("mouse click number:" + clicks);
44.     }
45.     public function gameOver():void {
46.         stage.removeEventListener(MouseEvent.CLICK, onMouseClickEvent);
47.         gameState = STATE_INIT;
48.         trace("game over");
49.     }
50. }
51.}
```

## 测试游戏

干嘛不去测试游戏呢？

如果你是 AS3 初学者并且你使用的开发工具是 Flash IDE。你可以通过下面的步骤来测试游戏。

- 1.把代码保存到 Game.as 文件中。
- 2.建立一个新的 Flash AS3 的 fla 文件 并命名为 clickgame.fla。保存。
- 3.把 as 文档类的名字输入到你新建的 fla 文档的类参数中。
- 4.修改舞台大小为 550\*400。

测试游戏吧！



如果你不是用 Flash IDE 开发的话，我们可以开始第二章了。也许，我们会再次用这种方法来测试本书中的其他游戏，所以如果你忘了，回来复习一下。

好玩么？好吧，反正肯定会赢。但是它是大游戏的基础，它包括了之前我们说的游戏框架的三个核心，和一些不是那么明显的珍贵体验。比如说，在 `gameOver` 函数中，我们移除了鼠标点击事件的监听。这就是一个清空事件监听的练习体验啊。在 AS3 中，一个非常常见的漏洞就是有的事件监听没有被移除。如果你忘了，你可能发现你的游戏运行的越来越慢。所以如果你一开始就养成这个好习惯的话，你会发现你的游戏制作生涯更加顺畅。

## 你的第二个游戏：扎气球

现在你的第一个游戏已经完成了。让我们沿用这个思路一鼓作气吧！你的第二个游戏，我把这个游戏称为扎气球：玩家控制一个旋转的刀刃去戳破你所能看到的每一个的气球。如果你漏掉了 5 个气球的话，你就 `GameOver` 了。我们将会在这个项目中扩展先前使用的游戏框架。

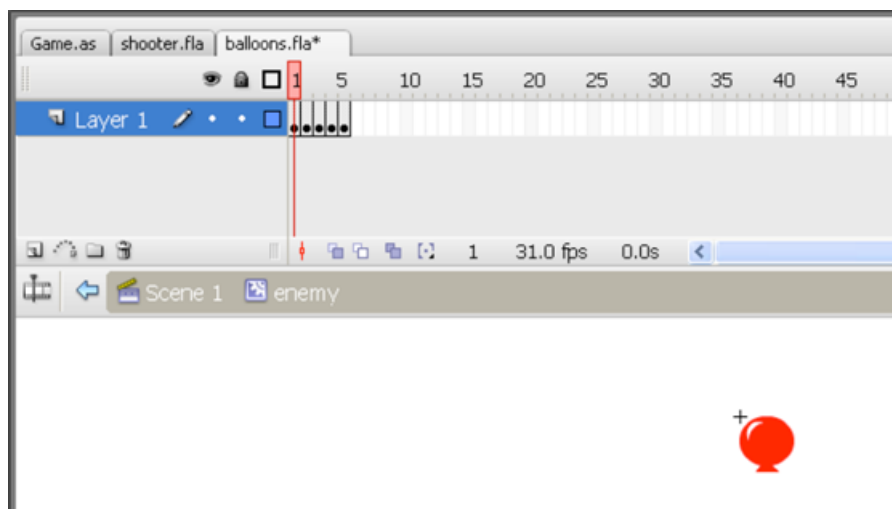
### 这个游戏的资源

在我们开始制作游戏之前需要给你的游戏创建一些资源。现在，来看看我们需要什么样资源。

#### 图形

在这个游戏中我们共需要三个图形资源，而且在元件库里都是影片剪辑。

- 敌人：这个游戏的“敌人”就是气球啦。这个影片剪辑一共有五帧，每一帧都是不同颜色的气球，分别是红，蓝，绿，黄，紫。图示 1-3 展示了这个剪辑。



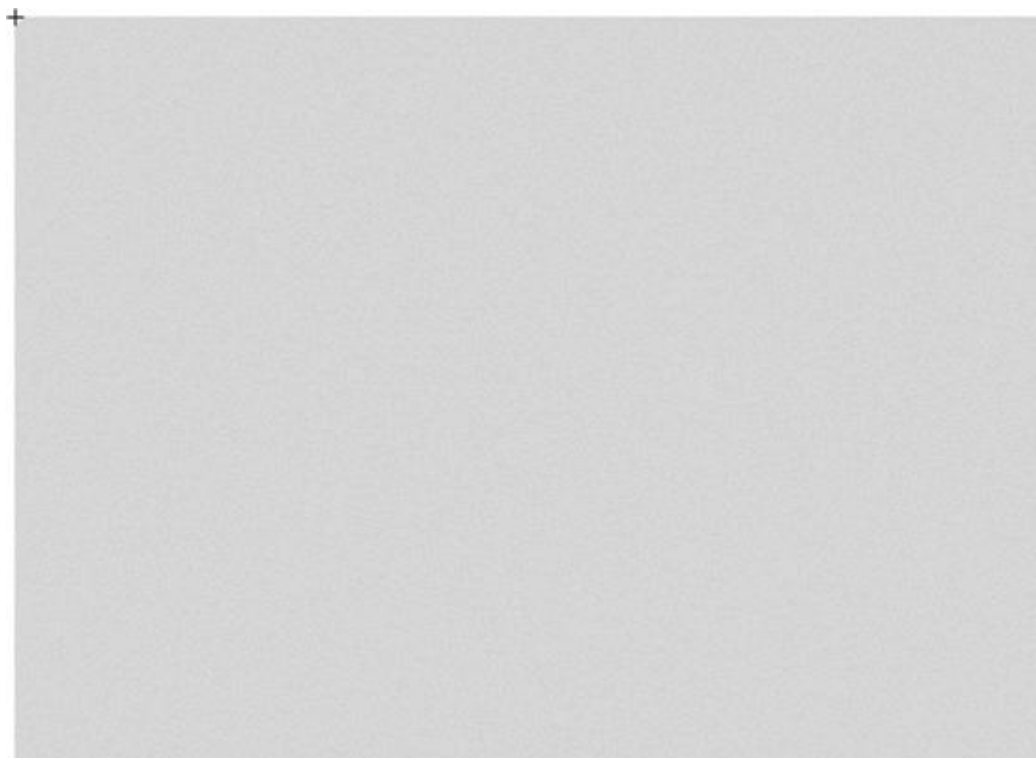
图示 1-3：气球的影片剪辑

- 玩家：玩家就是一个静止的刀刃（图示 1-4）。它是一个静态的图形，但是我们会用代码来让它转起来。我们让这个图形的中心与影片剪辑的注册点重合了，这样旋转的时候就不会出现偏移。



图示 1-4. 玩家的影片剪辑

- **背景：**我们要建立一个 550\_400 的纯色图片作为背景（图示 1-5）。我们使用图片作为背景有两个理由：第一，你在 Flash 中设置的背景色可能被 Flash IDE 修改，而且在网页上的显示也不怎么好。这会让你的游戏陷入困境。第二，一个背景图片可以更灵活的运用一些特殊的效果。



图示 1-5. 背景影片剪辑

## 声音

我们只需要一个声音资源就够了。在库里我们把它命名为 POP，让刀刃碰到气球的时候就发出噗的一声。

## 设置

这个游戏需要在你的 AS3 工程中进行如下设置。

- 尺寸：550\_400
- 帧频：30

## 代码

现在你库里图形和声音资源都有了，就可以创建我们的扎气球游戏了。



在这个课程中，我们只需要一个 Game.as 文档类在我们的 Flash 项目中就可以了，像我们先前做的那样，这个游戏的代码精简到最少了，我们会给你看全部的代码，并给你描述它跟第一个游戏的代码有哪些明显的不同。

```
package {  
    import flash.display.MovieClip;  
    import flash.events.Event;  
    import flash.events.MouseEvent;  
    import flash.display.*;  
    import flash.events.*;  
    import flash.geom.Rectangle;  
    import flash.media.Sound;  
    import flash.text.*;  
    public class Game extends flash.display.MovieClip{  
        public static const STATE_INIT:int = 10;  
        public static const STATE_PLAY:int = 20;  
        public static const STATE_END_GAME:int = 30;  
        public var gameState:int = 0;  
        public var score:int = 0;  
        public var chances:int = 0;  
        public var bg:MovieClip;  
        public var enemies:Array;  
        public var player:MovieClip;  
        public var level:Number = 0;  
        public var scoreLabel:TextField = new TextField();  
        public var levelLabel:TextField = new TextField();  
        public var chancesLabel:TextField = new TextField();  
        public var scoreText:TextField = new TextField();  
        public var levelText:TextField = new TextField();  
        public var chancesText:TextField = new TextField();  
        public const SCOREBOARD_Y:Number = 380;  
        public function Game() {  
            addEventListener(Event.ENTER_FRAME, gameLoop);  
            bg = new BackImage();  
            addChild(bg);  
            scoreLabel.text = "Score:";  
        }  
    }  
}
```



```
levelLabel.text = "Level:";
chancesLabel.text = "Misses:"
scoreText.text = "0";
levelText.text = "1";
chancesText.text = "0";
scoreLabel.y = SCOREBOARD_Y;
levelLabel.y = SCOREBOARD_Y;
chancesLabel.y = SCOREBOARD_Y;
scoreText.y = SCOREBOARD_Y;
levelText.y = SCOREBOARD_Y;
chancesText.y = SCOREBOARD_Y;
scoreLabel.x = 5;
scoreText.x = 50;
chancesLabel.x = 105;
chancesText.x = 155;
levelLabel.x = 205;
levelText.x = 260;
addChild(scoreLabel);
addChild(levelLabel);
addChild(chancesLabel);
addChild(scoreText);
addChild(levelText);
addChild(chancesText);
gameState = STATE_INIT;
}

public function gameLoop(e:Event):void {
    switch(gameState) {
        case STATE_INIT :
            initGame();
            break;
        case STATE_PLAY:
            playGame();
            break;
        case STATE_END_GAME:
```



```
        endGame();
        break;
    }
}

public function initGame():void {
    score = 0;
    chances = 0;
    player = new PlayerImage();
    enemies = new Array();
    level = 1;
    levelText.text = level.toString();
    addChild(player);
    player.startDrag(true,new Rectangle(0,0,550,400));
    gameState = STATE_PLAY;
}

public function playGame():void {
    player.rotation += 15;
    makeEnemies();
    moveEnemies();
    testCollisions();
    testForEnd();
}

public function makeEnemies():void {
    var chance:Number = Math.floor(Math.random() *100);
    var tempEnemy:MovieClip;
    if (chance < 2 + level) {
        tempEnemy = new EnemyImage()
        tempEnemy.speed = 3 + level;
        tempEnemy.gotoAndStop(Math.floor(Math.random()*5) + 1);
        tempEnemy.y = 435;
        tempEnemy.x = Math.floor(Math.random() * 515)
        addChild(tempEnemy);
        enemies.push(tempEnemy);
    }
}
```



```
public function moveEnemies():void {
    var tempEnemy:MovieClip;
    for (var i:int = enemies.length - 1; i >= 0; i--) {
        tempEnemy = enemies[i];
        tempEnemy.y -= tempEnemy.speed;
        if (tempEnemy.y < -35) {
            chances++;
            chancesText.text = chances.toString();
            enemies.splice(i, 1);
            removeChild(tempEnemy);
        }
    }
}

public function testCollisions():void {
    var sound:Sound = new Pop();
    var tempEnemy:MovieClip;
    for (var i:int = enemies.length - 1; i >= 0; i--) {
        tempEnemy = enemies[i];
        if (tempEnemy.hitTestObject(player)) {
            score++;
            scoreText.text = score.toString();
            sound.play();
            enemies.splice(i, 1);
            removeChild(tempEnemy);
        }
    }
}

public function testForEnd():void {
    if (chances >= 5) {
        gameState = STATE_END_GAME;
    } else if (score > level * 20) {
        level++;
        levelText.text = level.toString();
    }
}
```

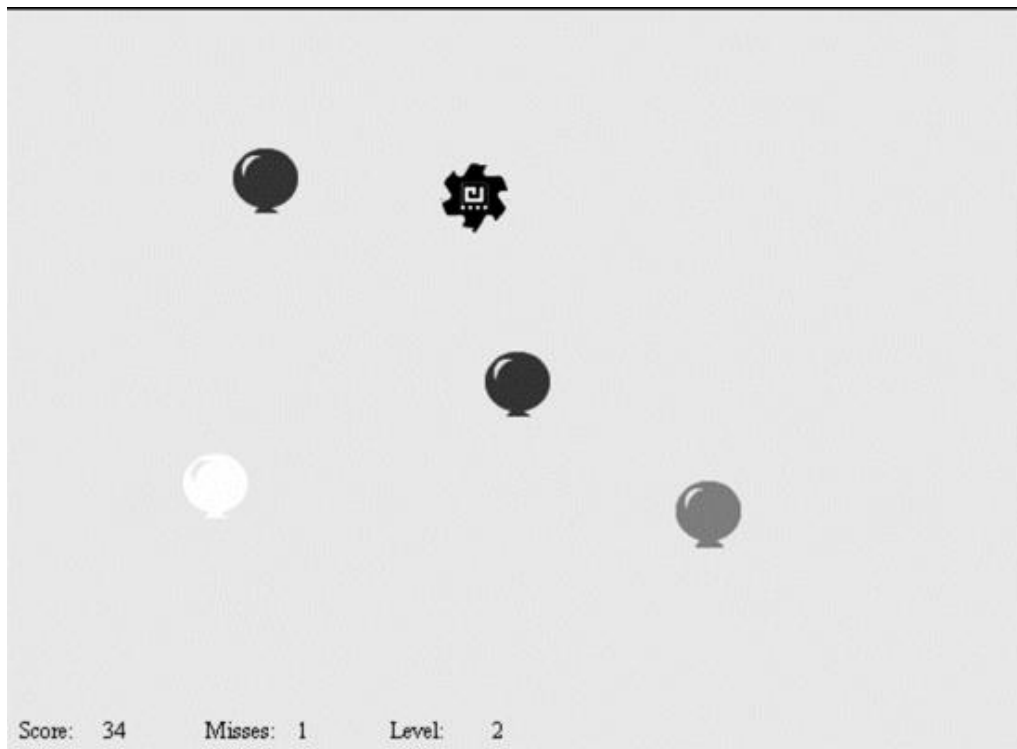


```
}  
  
public function endGame():void {  
    for(var i:int = 0; i < enemies.length; i++) {  
        removeChild(enemies[i]);  
    }  
    enemies = [];  
    player.stopDrag();  
}  
}  
}
```

好了，你被耍了，这个新代码看起来很多，一点也不小。但是你别以为 AS3 游戏是那么的简单的。实际上，这个示例并不比第一个游戏复杂多少。我们仍然是通过调用一个游戏循环等待用户输入操作。我们只是加了一些修饰使得它更像一个真正的 Flash 游戏罢了。

## 扎气球游戏测试

这个就不阐述了，先前的游戏测试方法应该能让你应用在这个游戏上了。



图示 1-6 扎气球游戏演示



在这个游戏中，玩家用鼠标控制旋转的刀刃。气球从屏幕的低端飘上来，玩家必须用这个刀刃去把所有出现的气球戳破。随着游戏的进行，等级会持续提升，气球飘动的速度也越来越快。如果玩家漏掉了五个气球，游戏就结束了。由于没有设立一个重置方法，所以要重新玩的话只能关掉它再打开。

## 分解扎气球游戏的代码

我们会在这部分里把扎气球游戏的代码进行分段说明，这有助于你更好的理解游戏的流程。但是对于不重要的部分我们不会讲述的太细致。

### 导入

首先，我们必须给我们的游戏导入必要的类。flash.geom.Rectangle 类是用来给鼠标创建一个矩形限制区的，flash.media.Sound 是用来播放气球爆破声音的，而 flash.text 则是用来给计分板上建立文本域的。

```
package {  
  
    import flash.display.MovieClip;  
  
    import flash.events.Event;  
  
    import flash.events.MouseEvent;  
  
    import flash.display.*;  
  
    import flash.events.*;  
  
    import flash.geom.Rectangle;  
  
    import flash.media.Sound;  
  
    import flash.text.*;
```

### 变量

现在我们来到游戏代码内部。第一个定义的变量就是玩家的分数(从第一个游戏中的 clicks 变量引申过来的)，还有 chances 变量（记录玩家漏掉了多少个气球），还有 level（等级）。

```
public class Game extends flash.display.MovieClip{  
  
    const STATE_INIT:int = 10;  
  
    const STATE_PLAY:int = 20;  
  
    const STATE_END_GAME:int = 30;  
  
    public var gameState:int = 0;  
  
    public var score:int = 0;  
  
    public var chances:int = 0;
```



```
public var level:Number = 0;
```

然后,我们还需要创建一些其他类型的变量来建立游戏需要的元素,bg 就是用来引用库中游戏背景图片的 ( BackImage ( ) )。enemies 就是一个存储敌人 ( 气球 ) 的数组,player 就是引用玩家的图片 ( PlayerImage ( ) ) 的。

```
public var bg:MovieClip;
```

```
public var enemies:Array;
```

```
public var player:MovieClip;
```

下面我们要给计分板创建一些文本域。有分数、等级、和机会 ( 漏网数 )。我们要设计一个自己的计分板,所以我们要给每个元素建立两个文本域,一个用来显示标题,一个用来显示内容。

```
public var scoreLabel:TextField = new TextField();
```

```
public var levelLabel:TextField = new TextField();
```

```
public var chancesLabel:TextField = new TextField();
```

```
public var scoreText:TextField = new TextField();
```

```
public var levelText:TextField = new TextField();
```

```
public var chancesText:TextField = new TextField();
```

最后,我们需要创建一个新的变量,我们需要给计分板定义一个固定的 Y 坐标,用来使计分板保持在屏幕的下端,既然舞台的高是 400,那么我们就把 y 坐标设定为 380 就好了。

```
public const SCOREBOARD_Y:Number = 380;
```

我们声明这个常量的原因就是为了让计分板需要调整的时候,里面的其他部件也跟着移动。至于 x 坐标嘛,因为每个部件的都不一样,所以我们之后再说。

## 构造函数

现在,我们来看看游戏的构造函数。首先要做的就是引用 BackImg 并用 addChild 把它加入到舞台上。我们不需要设置它的坐标。因为默认的就是在 0,0 点,那就是我们需要的。

```
public function Game() {
```

```
addEventListener(Event.ENTER_FRAME, gameLoop);
```

```
bg = new BackImage();
```

```
addChild(bg);
```

然后,我们要建立计分板,不走运的是,在 AS3 程序中,建立文本的代码比我们想的多一点。比起设置计分板里的变量来,这似乎挺让人难受的。我们需要给每一个文本域设置 x 坐标,而 y 坐标嘛,想想我们之前声明的 SCOREBOARD\_Y 吧。坐标都设好后,我们就通过 addChild ( TextFeild ) 方法来把它们加



入到显示列表中去，虽然在 Flash IDE 中，我们可以很方便的把这些东西拖拽到舞台上，但是用代码有个非常大的好处，它可以被很容易的转移到 Flex 中或者其它语言里。

```
scoreLabel.text = "Score:";

levelLabel.text = "Level:";

chancesLabel.text = "Misses:"

scoreText.text = "0";

levelText.text = "1";

chancesText.text = "0";

scoreLabel.y = SCOREBOARD_Y;

levelLabel.y = SCOREBOARD_Y;

chancesLabel.y = SCOREBOARD_Y;

scoreText.y = SCOREBOARD_Y;

levelText.y = SCOREBOARD_Y;

chancesText.y = SCOREBOARD_Y;

scoreLabel.x = 5;

scoreText.x = 50;

chancesLabel.x = 105;

chancesText.x = 155;

levelLabel.x = 205;

levelText.x = 260;

addChild(scoreLabel);

addChild(levelLabel);

addChild(chancesLabel);

addChild(scoreText);

addChild(levelText);

addChild(chancesText);

gameState = STATE_INIT;

}
```



## 初始化游戏循环

既然我们在构造函数的最后把游戏状态设置为 STATE\_INIT 了，那么你就应该知道 initGame ( ) 这个函数将会被 gameLoop ( ) 函数调用了。initGame ( ) 函数重置了我们开始游戏必备的所有变量的初始值。包括分数，等级，和机会（漏网数）。并且把等级这个变量字符串化以便赋给 levelText.text ( 因为 levelText.text 的属性值必须是字符串 )。

下面，我们要创建一个 PlayerImage() 去从库里引用玩家的图形（就是那个刀刃）。然后用 addChild ( player ) 方法添加到显示列表。既然这个图形是用鼠标控制的，那我们就要用这种方式了：使用鼠标拖拽来控制元件。我们通过 player.startDrag() 函数来实现，它有两个参数。

- **lockCenter**：这是是否将组件锁定到鼠标指针的中央，因为我们需要这么做，所以我们把它设置为真。
- **bounds**：定义一个矩形（通过 flash.geom.Rectangle 类）来限制鼠标可拖拽的范围。这个参数可以不填，但是我们要加上它，因为我们不想让鼠标为所欲为的拖拽，如果你以后需要让元件只能在 x 轴上移动来让游戏更有难度。你可以把这个参数设置为 0, 0, 550, 0

最后我们创建一个 enemies 数组来存储气球们 ( EnemyImage )。

```
public function initGame() {  
  
    score = 0;  
  
    chances = 0;  
  
    level = 1;  
  
    levelText.text = level.toString();  
  
    player = new PlayerImage();  
  
    addChild(player);  
  
    player.startDrag(true, new Rectangle(0,0,550,400));  
  
    enemies = new Array();  
  
    gameState = STATE_PLAY;  
  
}
```

## 开始游戏函数

playGame ( ) 函数将会被 gameLoop ( ) 函数调用，gameLoop ( ) 是负责调整游戏状态的循环函数，而 playGame ( ) 的第一件事就是让玩家控制的元件每次旋转 15 度，这使得它看起来就像旋转的刀刃一样。在 AS3 中通过调整影片剪辑的 rotation 属性来实现旋转是非常普遍的方法。rotation 就是角度。正数的时候表示顺时针，负数就是逆时针。



然后调用的是 `makeEnemies()` 函数去让你的敌人出现，`moveEnemies()` 函数负责更新气球的 y 坐标以及在气球飘出屏幕的时候移除它们。`testCollisions()` 函数是检测玩家是否与它的敌人发生碰撞了，如果是的话就触发一个事件。最后 `testForEnd()` 函数是用来查看等级 (level) 是不是该增加了或者游戏是不是该结束了。这些方法在下面的代码中被展示。

```
public function playGame() {  
    player.rotation += 15;  
    makeEnemies();  
    moveEnemies();  
    testCollisions();  
    testForEnd();  
}
```

## 创建气球

在许多游戏里，你会需要一些不需要玩家进行交互就能调用的方法。在动作游戏里比如说我们的这个扎气球游戏。这个方法的核心就是创建玩家必须面对的敌人。包括出现时间、初始数值，和这些敌人的位置。是这些因素使得这类游戏充满了挑战和趣味。有许多的方式来进行创建这些东西。通常都是使用随机数和精确的定位。然而，在本书中，我们会介绍多种不同的方法以适合不同的游戏。

`makeEnemies()` 函数在每一帧进行检测是否需要创建一个气球，气球出现的频率是通过一个随机的数值获得的，这个随机数与游戏当前的 level 等级有关。首先，我们通过调用 `Math.floor(Math.random()*100)` 来获取一个 0 到 99 之间的随机数 然后把这个随机数赋给 `chance` 变量。接下来，我们就要检测这个 `chance` 变量是不是比 `level` 变量的值加 2 还小（示例：等级 1 的时候是 3，等级 2 的时候是 4，依次类推）。随着等级的增加，气球出现的几率也随之增加。

当条件成立的时候，就是创建一个气球并把它放到舞台上的时候了。我们是通过声明一个名为 `tempEnemy` 的临时变量来创建的。首先，我们引用 `EnemyImage()` 并把它赋给 `tempEnemy` 变量。然后我们通过动态变量设置它的速度。Flash 的影片剪辑可以随时通过声明一个变量来创建。虽然这是很灵活的方式，但是代价就是牺牲更多的内存。我们现在使用这种方法是因为在这个游戏中这么做起来很简单，而且我们不必单独写一个 `Enemy` 的类了（这些东西我们在书中的后面会讲到）。我们通过一个动态变量（`level` 数值+3）来设置它的速度。这意味着随着等级的提升，气球的移动速度会越来越快，这会给游戏增加点挑战性。

我们的 `EnemyImage` 影片剪辑有 5 帧，每一帧包含了不同颜色的气球。我们用随机数（`Math.floor(Math.random()*5+1)`）和 `tempEnemy.gotoAndStop` 的方式来随机选取其中一种颜色。虽然这些颜色只是为了游戏的视觉效果，但是在更复杂的游戏规则中，不同颜色的气球可以被用于不同的分值奖励。

然后，我们给这个气球初始化它的初始坐标值。Y 坐标总是 435，表示从屏幕的底端开始移动。X 坐标是一个 0 到 514 的随机数。这个气球元件的宽度是 35 像素，而且注册点是 0，0。这种设置会让气球在屏



幕的范围内出现，如果我们把  $x$  的取值范围扩大到 550（舞台宽度），有些气球就可能有一部分显示在屏幕外面。这会让玩家不可避免的遗漏掉一些气球，那样很不公平。最后我们用 `addChild()` 方法把这个气球加入到显示列表，并把它压入到敌人数组（`enemies array`）中以便跟踪。

```
public function makeEnemies() {  
  
    var chance:Number = Math.floor(Math.random() *100);  
  
    if (chance < 2 + level) {  
  
        var tempEnemy:MovieClip;  
  
        tempEnemy = new EnemyImage()  
  
        tempEnemy.speed = 3 + level;  
  
        tempEnemy.gotoAndStop(Math.floor(Math.random()*5)+1);  
  
        tempEnemy.y = 435;  
  
        tempEnemy.x = Math.floor(Math.random()*515)  
  
        addChild(tempEnemy);  
  
        enemies.push(tempEnemy);  
  
    }  
  
}
```

## 移动气球

动作游戏的另一个核心函数就是移动屏幕上的显示对象。我们要建立一个函数控制游戏中的所有显示对象移动(除了玩家用鼠标控制的对象)。这个游戏中只需要设置一个对象的移动就好了（气球），但是在单人游戏中，这种方法也可以被用来控制其他类型的对象。

`moveEnemies()` 函数遍历 `enemies` 数组，更新每一个气球的  $y$  坐标。既然是要往屏幕上方移动，我们就减少每一个气球的  $y$  值。如果一个气球的  $y$  属性小于 -35 了，我们就把它从 `enemies` 数组中用 `removeChild()` 方法移除。注意这个循环的查询方式（从 `enemies.length-1` 到 0）。在本书中你可以看见这种循环的遍历方式很多次。从数组尾端进行遍历是游戏程序中一个非常有用的技巧，尤其是你要移除数组中的元素时，如果你从数组的前面开始遍历，那么循环很可能出现错误，因为数组的长度会随着你移除它的元素而改变。

```
public function moveEnemies() {  
  
    var tempEnemy:MovieClip;  
  
    for (var i:int = enemies.length-1;i >= 0;i--) {  
  
        tempEnemy = enemies[i];
```



```
tempEnemy.y -= tempEnemy.speed;

if (tempEnemy.y < -35) {

    chances++;

    chancesText.text = chances.toString();

    enemies.splice(i,1);

    removeChild(tempEnemy);

}

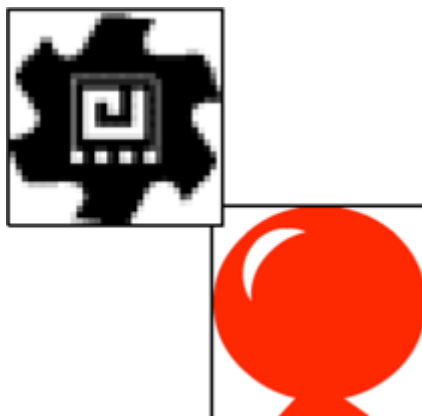
}
```

## 碰撞检测

现在，我们来看看这个游戏的灵魂——如何检测两个对象之间是否碰撞。许多游戏的核心就是碰撞检测，它是用来检测两个对象之间是否发生碰触的，如果是的话就给对象做一些事情。碰撞检测可不是一个简单的算法，它是由整个类文件算法实现的以便应付不同的情况。在这个游戏里，我们会使用 AS3 中一个非常简单的办法。

说到这里，我有一个好消息和一个坏消息。好消息是 AS3 有着跟它的旧版本（AS2）同样的方法：为影片剪辑用的 `hitTest()` 方法。现在改叫 `MovieClip.hitTestObject()` 方法了，但是它跟以前一样好使。你只需要传递另一个影片剪辑就能知道它们是不是碰撞了。

在这个游戏中，我们检测一个气球（由 `tempEnemy` 定义的）是不是碰到了玩家：`tempEnemy.hitTestObject(player)`。像先前说的那样，`hitTestObject()` 既有好处也有坏处，好的是使用起来真的很方便，坏的就是它在游戏中不太合适。`hitTestObject()` 方法使用矩形碰撞检测原理，这意味着任何对象都被一个看不见的矩形包裹住，无论这个对象本身是不是一个矩形，用这个方法检测的时候都会把它当作一个矩形来检测，这样的话，有时你明明看见一个对象那里明明没有任何图像却与另一个对象碰撞了，图示 1-7 表示了这种情况。



图示 1-7. 矩形碰撞检测（矩形的边界是后画的，其实看不见的）



对于矩形对象来说，矩形碰撞检测是没有任何问题的，但是对于一个圆形、丁字形或者其它形状的时候就不好用了。我们之所以在这个游戏中使用它是因为这简单快捷，但是在书中的后面部分，我们会给你介绍很多更精确的碰撞检测（包括像素级别碰撞检测）。顺便说一下，MovieClip.hitTestPoint()方法可以检测一个对象是否碰到一个点，但是我们不讲它，让你自己研究去。

这个函数从 enemies 数组的尾部开始检测，看看是否有跟玩家碰撞的气球。如果有，我们就把分数加 1，并且更新文本域里的值。然后通过 enemies.splice(I,1)方法从数组中移除这个气球，再用 removeChild ( ) 方法把这个气球从显示列表上移除。

还有一个有意思的事就是在这个函数里我们还要播放声音。我们创建一个库中 Pop 声音的引用，这个写作 sound.play()的方法让气球每次破了的时候就发出“pop”的一声。

```
public function testCollisions() {  
    var sound:Sound = new Pop();  
    var tempEnemy:MovieClip;  
    for (var i:int=enemies.length-1;i>=0;i--) {  
        tempEnemy = enemies[i];  
        if (tempEnemy.hitTestObject(player)) {  
            score++;  
            scoreText.text = score.toString();  
            sound.play();  
            enemies.splice(i,1);  
            removeChild(tempEnemy);  
        }  
    }  
}
```

## 结束游戏

我们现在完成了大部分的代码，只有两个事情还没提。一个就是 testForEnd ( ) 函数，这个函数被 gameLoop ( ) 函数调用并检测当前 level 等级或者机会的剩余。首先，它看看游戏是不是该玩完了。我们检测 chances 变量是不是大于 5 了，如果是的话，我们就把游戏状态改为 STATE\_END\_GAME，然后 gameLoop 就会调用 endgame ( ) 函数了。如果玩家还没有那么惨，那么我们就看看是不是该增加一下 level 等级来让游戏更难一点。Level 的升级规则是分数是不是比当前等级 level\*20 的值大。既然每个气球都是一分，那么我们很容易判断出什么时候该提升游戏等级了。



```
public function testForEnd() {  
    if (chances >= 5) {  
        gameState = STATE_END_GAME;  
    } else if(score > level * 20) {  
        level ++;  
        levelText.text = level.toString();  
    }  
}
```

最后，在游戏状态是 STATE\_END\_GAME 的时候，endGame ( ) 函数就被调用了。这个函数里移除屏幕上所有的敌人然后保持静止。你可能想知道如何重新开始游戏，别着急，跟着本书学习，你会很容易地学习如何开始游戏、结束游戏并改变等级。

```
public function endGame() {  
    for(var i:int = 0; i < enemies.length; i++) {  
        removeChild(enemies[i]);  
        enemies = [];  
    }  
    player.stopDrag();  
}
```

就这样吧，扎气球游戏虽然是一个相当简单的游戏，但是，一个状态循环机制，一个多对象处理和用户输入处理，这些是你更深入探索 AS3 游戏的好起点

## 创建你的第三个游戏：像素射手

在我们进行第二章来制作更炫更复杂的游戏之前。为什么不来试试在扎气球游戏的基础上建立另一种类型的游戏呢？我们不会用扎气球游戏那种详尽的说明方法，而是重点讲述它与众不同的部分，其它的由你自己来摸索。这个游戏的代码是由扎气球游戏代码改进而成的。这说明我们的第二游戏说和保持进步的理论是正确的。

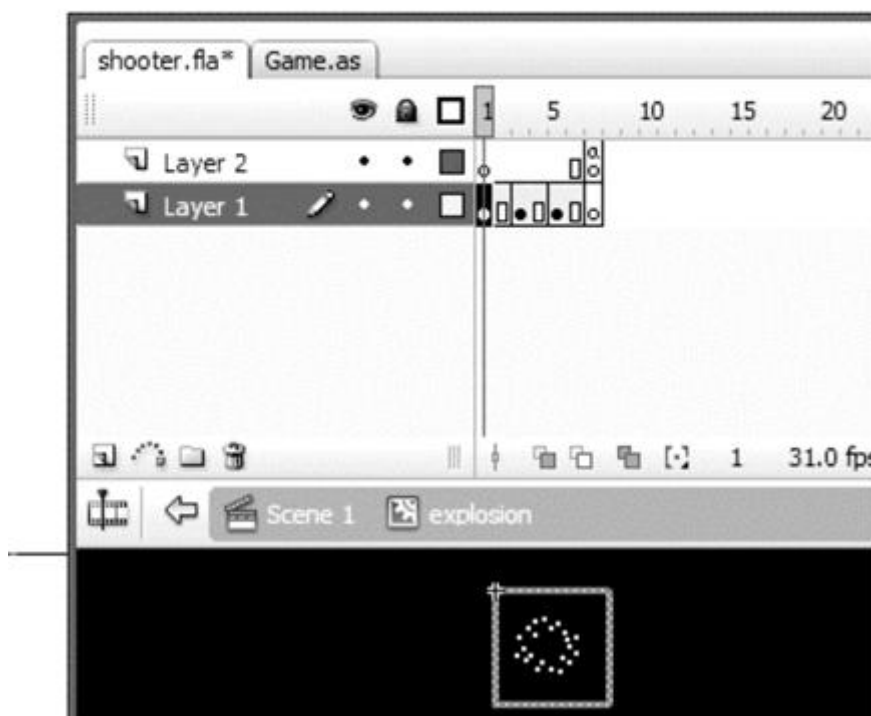
### 游戏设计

像素射手是一个非常平凡的射击类游戏。敌人从屏幕顶端出现，玩家的任務就是消灭它们。玩家用鼠标控制飞船移动，用左键单击发射导弹。敌人的速度会越来越快，当玩家撞到敌人的飞机时，玩家的飞机爆炸并减少一条生命。



## 图形资源

在这个游戏中我们有四个影片剪辑，每一个都放在资源库里。第一个就是一个名为 ExplodeImage 的爆炸效果，它的导出成一个类。在它的三帧中，每一个都绘制了一个单独的 gif 图片。每两个图片的变化构成了一个动画效果。在最后一帧上我们加上了 stop ( ) 命令。这就表示影片剪辑在播放到第七帧时停止。这有助于告诉我们该把它移除了。



图示 1-8. 爆炸效果影片剪辑展示

另外三个图形在图示 1-9 中展示出来了：

- PlayerImage：玩家控制的飞机
- EnemyImage：玩家要消灭的敌人
- MissileImage：玩家发射的导弹



图示 1-9. PlayerImage, EnemyImage, MissileImage 影片剪辑



## 声音资源

我们在这里需要两个声音，存放在资源库里并导出为类。它们是：

- Explode：飞机爆炸时的声音效果
- Shoot：发射导弹的声音效果

## 代码

下面就是全部的代码了。我们是在扎气球游戏的代码基础上建立的。（译者注：原文中把与扎气球游戏代码不同的部分加粗了）

```
package {  
  
    import flash.display.MovieClip;  
  
    import flash.events.Event;  
  
    import flash.events.MouseEvent;  
  
    import flash.display.*;  
  
    import flash.events.*;  
  
    import flash.geom.Rectangle;  
  
    import flash.media.Sound;  
  
    import flash.text.*;  
  
    public class Game extends flash.display.MovieClip{  
  
        public static const STATE_INIT:int = 10;  
  
        public static const STATE_START_PLAYER:int = 20;  
  
        public static const STATE_PLAY_GAME:int = 30;  
  
        public static const STATE_REMOVE_PLAYER:int = 40;  
  
        public static const STATE_END_GAME:int = 50;  
  
        public var gameState:int = 0;  
  
        public var score:int = 0;  
  
        public var chances:int = 0;  
  
        public var bg:MovieClip;  
  
        public var enemies:Array;  
  
        public var missiles:Array;
```



```
public var explosions:Array;
```

```
public var player:MovieClip;
```

```
public var level:Number = 0;
```

```
public var scoreLabel:TextField = new TextField();
```

```
public var levelLabel:TextField = new TextField();
```

```
public var chancesLabel:TextField = new TextField();
```

```
public var scoreText:TextField = new TextField();
```

```
public var levelText:TextField = new TextField();
```

```
public var chancesText:TextField = new TextField();
```

```
public const SCOREBOARD_Y:Number = 5;
```

```
public function Game() {
```

```
    addEventListener(Event.ENTER_FRAME, gameLoop);
```

```
    bg = new BackImage();
```

```
    addChild(bg);
```

```
    scoreLabel.text = "Score:";
```

```
    levelLabel.text = "Level:";
```

```
chancesLabel.text = "Ships:";
```

```
    scoreText.text = "0";
```

```
    levelText.text = "1";
```

```
    chancesText.text = "0";
```

```
    scoreLabel.y = SCOREBOARD_Y;
```

```
    levelLabel.y = SCOREBOARD_Y;
```

```
    chancesLabel.y = SCOREBOARD_Y;
```

```
    scoreText.y = SCOREBOARD_Y;
```

```
    levelText.y = SCOREBOARD_Y;
```

```
    chancesText.y = SCOREBOARD_Y;
```

```
    scoreLabel.x = 5;
```

```
    scoreText.x = 50;
```



```
chancesLabel.x = 105;

chancesText.x = 155;

levelLabel.x = 205;

levelText.x = 260;

scoreLabel.textColor = 0xFF0000;

scoreText.textColor = 0xFFFFFF;

levelLabel.textColor = 0xFF0000 ;

levelText.textColor = 0xFFFFFF;

chancesLabel.textColor = 0xFF0000;

chancesText.textColor= 0xFFFFFF;

addChild(scoreLabel);

addChild(levelLabel);

addChild(chancesLabel);

addChild(scoreText);

addChild(levelText);

addChild(chancesText);

gameState = STATE_INIT;
}

public function gameLoop(e:Event) {

    switch(gameState) {

        case STATE_INIT :

            initGame();

            break

        case STATE_START_PLAYER:

            startPlayer();

            break;

        case STATE_PLAY_GAME:
```



```
        playGame();

        break;

        case STATE_REMOVE_PLAYER:

            removePlayer();

            break;

        case STATE_END_GAME:

            break;

    }

}

public function initGame() {

    stage.addEventListener(MouseEvent.CLICK, onMouseDownEvent);

    score = 0;

    chances = 3;

    enemies = new Array();

    missiles = new Array();

    explosions = new Array();

    level = 1;

    levelText.text = level.toString();

    player = new PlayerImage();

    gameState = STATE_START_PLAYER;

}

public function startPlayer() {

    addChild(player);

    player.startDrag(true, new Rectangle(0, 365, 550, 365));

    gameState = STATE_PLAY_GAME;

}

public function removePlayer() {

    for(var i:int = enemies.length-1; i >= 0; i--) {
```



```
        removeEnemy(i);
    }
    for(i = missiles.length-1; i >= 0; i--) {
        removeMissile(i);
    }
    removeChild(player);
    gameState = STATE_START_PLAYER;
}

public function playGame() {
    makeEnemies();
    moveEnemies();
    testCollisions();
    testForEnd();
}

public function makeEnemies() {
    var chance:int = Math.floor(Math.random() *100);
    var tempEnemy:MovieClip;
    if (chance < 2 + level) {
        tempEnemy = new EnemyImage()
        tempEnemy.speed = 1 + level;
        tempEnemy.y = -25;
        tempEnemy.x = Math.floor(Math.random()*515)
        addChild(tempEnemy);
        enemies.push(tempEnemy);
    }
}

public function moveEnemies() {
```



```
var tempEnemy:MovieClip;

for (var i:int = enemies.length-1;i>=0;i--) {
    tempEnemy = enemies[i];
    tempEnemy.y+=tempEnemy.speed;
    if (tempEnemy.y > 435) {
        removeEnemy(i);
    }
}

var tempMissile:MovieClip;

for (i=missiles.length-1;i>=0;i--) {
    tempMissile = missiles[i];
    tempMissile.y-=tempMissile.speed;
    if (tempMissile.y < -35) {
        removeMissile(i);
    }
}

var tempExplosion:MovieClip;

for (i=explosions.length-1;i>=0;i--) {
    tempExplosion = explosions[i];
    if (tempExplosion.currentFrame >= tempExplosion.totalFrames) {
        removeExplosion(i);
    }
}

}

public function testCollisions() {
    var tempEnemy:MovieClip;
    var tempMissile:MovieClip;
```



```
enemy: for (var i:int=enemies.length-1;i>=0;i--) {  
    tempEnemy = enemies[i];  
    for (var j:int = missiles.length-1;j>=0;j--) {  
        tempMissile = missiles[j];  
        if (tempEnemy.hitTestObject(tempMissile)) {  
            score++;  
            scoreText.text = score.toString();  
            makeExplosion(tempEnemy.x, tempEnemy.y);  
            removeEnemy(i);  
            removeMissile(j);  
            break enemy;  
        }  
    }  
}  
  
for (i=enemies.length-1;i>=0;i--) {  
    tempEnemy = enemies[i];  
    if (tempEnemy.hitTestObject(player)) {  
        chances--;  
        chancesText.text = chances.toString();  
        makeExplosion(tempEnemy.x, tempEnemy.y);  
        gameState = STATE_REMOVE_PLAYER;  
    }  
}  
  
}  
  
public function makeExplosion(ex:Number, ey:Number) {
```



```
var tempExplosion:MovieClip;

tempExplosion = new ExplosionImage();

tempExplosion.x = ex;

tempExplosion.y = ey;

addChild(tempExplosion);

explosions.push(tempExplosion);

var sound:Sound = new Explode();

sound.play();

}

public function testForEnd() {

    if (chances <= 0) {

        removePlayer();

        gameState = STATE_END_GAME;

    } else if(score > level*30) {

        level++;

        levelText.text = level.toString();

    }

}

public function removeEnemy(idx:int) {

    removeChild(enemies[idx]);

    enemies.splice(idx,1);
```



```
}

public function removeMissile(idx:int) {

    removeChild(missiles[idx]);

    missiles.splice(idx,1);

}

public function removeExplosion(idx:int) {

    removeChild(explosions[idx]);

    explosions.splice(idx,1);

}

public function onMouseDownEvent(e:MouseEvent) {

    if (gameState == STATE_PLAY_GAME) {

        var tempMissile:MovieClip = new MissileImage();

        tempMissile.x = player.x + (player.width/2);

        tempMissile.y = player.y;

        tempMissile.speed = 5;

        missiles.push(tempMissile);

        addChild(tempMissile);

        var sound:Sound = new Shoot();

        sound.play();

    }

}

}
```



```
}
```

测试方法就不罗嗦了。

下面是游戏图片。



图示 1-10. 游戏展示

像素射手是一个简单又困难的游戏，敌人会在玩家面前一波一波的出现，而且越来越快。玩家用鼠标控制飞机移动并用左键单机发射导弹，当导弹击中敌人的时候，我们就让它像扎气球里的敌人那样消失，然后播放一个爆炸的效果。

像这种类型的游戏，我们需要多增加两个游戏元素。除了敌人，我们还必须加入导弹的移动和爆炸效果。而除了检测玩家和敌人的碰撞，我们还要检测导弹和敌人的碰撞。这时如果玩家和敌人碰上了可就不是加分了，而是丢掉一个生命。分数的增加是通过导弹碰撞敌人触发的。也不会记录导弹的发射数，而是要记录玩家摧毁的敌人数量。

## 代码重点



---

我们不会逐行解说这个游戏的代码，但是我们会谈论这个代码的重点部分。

## 新变量

我们不需要增加很多的变量就能把扎气球游戏变成像素射手游戏，只需要把原来的变量变化一下就行了。

前两个新变量是表示两个游戏状态。是用来帮助重置玩家的。在扎气球游戏中，玩家不会被摧毁，所以我们不用重置它。然而这里的玩家是可以被摧毁的，所以我们需要用 STATE\_START\_PLAYER 和 STATE\_REMOVE\_PLAYER 变量来重置玩家。

```
const STATE_START_PLAYER:int = 20;
```

```
const STATE_REMOVE_PLAYER:int = 40;
```

下面的两个变量都是数组，它们会存储游戏中的新类型的对象：导弹（被玩家发射的）和爆炸特效（当敌人的飞船被导弹击中时的效果）。

```
public var missiles:Array;
```

```
public var explosions:Array;
```

## 重置玩家

既然我们需要重置玩家的飞船，我们就需要在游戏循环中给其加入分支句柄。在 STATE\_START\_PLAYER 状态下，我们要调用一个新的函数叫 startPlayer()。在 STATE\_REMOVE\_PLAYER 状态中，我们就需要调用 removePlayer() 函数。

```
public function gameLoop(e:Event) {
```

```
    switch(gameState) {
```



```
case STATE_INIT :
```

```
    initGame();
```

```
    break
```

```
case STATE_START_PLAYER:
```

```
    startPlayer();
```

```
    break;
```

```
case STATE_PLAY_GAME:
```

```
    playGame();
```

```
    break;
```

```
case STATE_REMOVE_PLAYER:
```

```
    removePlayer();
```

```
    break;
```

```
case STATE_END_GAME:
```

```
    break;
```

```
}
```

```
}
```

startPlayer ( ) 函数用 addChild ( ) 方法把玩家的对象加入到显示列表中，并让它与鼠标的拖拽做关联。不同的地方是，startDrag 的矩形参数需要限定在屏幕的低端，就跟所有的竖版射击游戏一样。

```
public function startPlayer() {
```

```
    addChild(player);
```

```
    player.startDrag(true,new Rectangle(0,365 ,550,365));
```



```
gameState = STATE_PLAY_GAME;

}
```

removePlayer ( ) 函数实际上要比你想像的复杂一点。既然我们要在玩家被击毁或者游戏结束时调用 removePlayer ( ) 函数，那么可以顺便移除屏幕上所有的敌人，导弹，和爆炸的特效。这就好像让游戏回到初始化状态一样。这样做有一个小缺陷，就是我们直接把玩家飞船从屏幕上移除后，然后就调用 ( starPlayer ( ) ( 通过设置游戏状态为 STATE\_START\_PLAYER ) 了，这样做会把玩家飞船重新放到屏幕上，但是我们没时间给上一个飞船做出一个爆炸效果了。不过对于这样第一个小游戏来说，这点小缺点还是可以接受的。我们现在不要在细节上考虑那么周全。但是在后面的课程中，我们会讨论如果创建细节来让你的游戏成为精美的作品。

```
public function removePlayer() {

    for(var i:int = enemies.length-1; i >= 0; i--) {

        removeEnemy(i);

    }

    for(i = missiles.length-1; i >= 0; i--) {

        removeMissile(i);

    }

    removeChild(player);

    gameState = STATE_START_PLAYER;

}
```

## 多对象跟踪

在扎气球游戏中，我们只移动了一种类型的对象，就是敌人的气球。而在像素射手中，我们需要移动三个对象。但是其实代码写起来都差不多。在本游戏中，敌人从上想下移动，所以我们要通过给它一个速



度值来更新它的 y 值，并在它到达屏幕底部时移除。导弹嘛就正好反过来，它就跟扎气球游戏中的气球一样：从下向上移动并在到达顶端的时候移除。

爆炸效果呢，实现起来稍微麻烦点。回忆一下爆炸效果的影片剪辑中的 7 帧，还有最后的那句 stop ( ) 命令。既然爆炸是不移动的，唯一要做的就是知道他们是不是播放到最后一帧了好把他们移除。既然所有的影片剪辑都有一个可检测的 currentFrame 属性和 totalFrames 属性（表示最后一帧），那么代码我们就能这么写：

```
if (tempExplosion.currentFrame >= tempExplosion.totalFrames){
```

下面是完整的对象跟踪代码：

```
public function moveEnemies() {  
  
    var tempEnemy:MovieClip;  
  
    for (var i:int = enemies.length-1;i>=0;i--) {  
  
        tempEnemy = enemies[i];  
  
        tempEnemy.y+=tempEnemy.speed;  
  
        if (tempEnemy.y > 435) {  
            removeEnemy(i);  
        }  
    }  
  
    var tempMissile:MovieClip;  
  
    for (i=missiles.length-1;i>=0;i--) {  
  
        tempMissile = missiles[i];  
  
        tempMissile.y-=tempMissile.speed;  
  
        if (tempMissile.y < -35) {
```



```
        removeMissile(i);

    }

}

var tempExplosion:MovieClip;

for (i=explosions.length-1;i>=0;i--) {

    tempExplosion = explosions[i];

    if (tempExplosion.currentFrame >= tempExplosion.totalFrames) {

        removeExplosion(i);

    }

}

}
```

## 发射导弹

发射导弹将会是像素射手的原创函数。既然玩家会用鼠标来控制发射，我们就需要给游戏添加一个事件侦听以便我们收到鼠标事件消息。我们在 `initGame()` 函数中加入这个监听。当鼠标点击的时候，我们就调用一个名为 `onMouseDownEvent()` 的函数。

```
public function initGame() {

    stage.addEventListener(MouseEvent.CLICK,onMouseDownEvent);

    ...

}
```

`onMouseDownEvent()` 函数是创建玩家发射的导弹的。首先要做的是检测游戏状态是不是 `STATE_PLAY_GAME`。因为这是导弹可以被发射的唯一状态。如果游戏状态的条件达成了，我们就创建一



个导弹对象，过程就像我们创建敌人对象并放到 enemies 数值中的方式一样，不过在这里，是放到一个 missiles 数组中。我们还要播放一个 Shoot ( ) 声音，让游戏玩起来更像那么回事。

```
public function onMouseDownEvent(e:MouseEvent) {
```

```
    if (gameState == STATE_PLAY_GAME) {
```

```
        var tempMissile:MovieClip = new MissileImage();
```

```
        tempMissile.x = player.x + (player.width/2);
```

```
        tempMissile.y = player.y;
```

```
        tempMissile.speed = 5;
```

```
        missiles.push(tempMissile);
```

```
        addChild(tempMissile);
```

```
        var sound:Sound = new Shoot();
```

```
        sound.play();
```

```
    }
```

```
}
```

## 多对象碰撞检测

在扎气球游戏汇中，我们要检测仅仅是玩家和 enemies 数组中对象的碰撞。而在像素射手中，我们需要两种碰撞情况：

敌人和导弹的

敌人和玩家的

我们需要在两个循环中去检测它们。第一个循环检测 enemies 数组和 missiles 数组之间的检测。首先，我们建立一个 for 循环向后遍历 enemies 数组，并检测每一个敌人和所有 missiles 数组中的所有导弹的碰撞。



为了检测碰撞，我们使用同样的方法 hitTestObject ( )：

```
if (tempEnemy.hitTestObject(tempMissile)) { }
```

如果我们检测到了一个碰撞发生了，我们就调用 createExplosion ( ) 函数 ( 要传递敌人的坐标以便知道我们该把爆炸特效放在哪里 )，更新分数，并移除发生碰撞的导弹和敌人。最后一行里的 break enemy；，是非常重要的。它的作用是跳出这个 enemy：for ( var i=enemies.length-1;i>=0;i-- ) 这个循环。这个 enemy：被叫做标识 ( Label )。我们经常用这个方法跳出一个循环碰撞检测。但是看起来会有些乱。这个标签方法 ( 或跳出方法 ) 可以让你在需要的时候跳出一个循环。我们需要这么做是因为我们要把这个 tempEnemy 和 tempMissile 从各自的数组中移除出去。如果我们不停止混换，我们就检测到一个不存在的值，AS3 就会报错。这个结构可能就像一个 “go to” 语句，但其实不一样。它更像是在说：“嘿，别循环我了，去检测上层循环吧。”

```
public function testCollisions() {  
  
    var tempEnemy:MovieClip;  
  
    var tempMissile:MovieClip;  
  
    enemy: for (var i:int=enemies.length-1;i>=0;i--) {  
  
        tempEnemy = enemies[i];  
  
        for (var j:int = missiles.length-1;j>=0;j--) {  
  
            tempMissile = missiles[j];  
  
            if (tempEnemy.hitTestObject(tempMissile)) {  
  
                score++;  
  
                scoreText.text = score.toString();  
  
                makeExplosion(tempEnemy.x, tempEnemy.y);  
  
                removeEnemy(i);  
  
                removeMissile(j);  
  
            }  
  
        }  
  
    }  
}
```



```
        breakenemy;

    }

}

}
```

然后还要检测玩家和敌人之间的碰撞。这个检测更像我们在扎气球中气球和玩家的那种方式。不同地方是，扎气球游戏里的分数增加是写在这个循环里的，而在这个游戏中，这个循环里要做的是把玩家的飞船摧毁。我们还需要传递一个事件来减少玩家的生命数并把游戏状态设为 STATE\_REMOVE\_PLAYER。

```
for (i=enemies.length-1;i>=0;i--) {

    tempEnemy = enemies[i];

    if (tempEnemy.hitTestObject(player)) {

        chances--;

        chancesText.text = chances.toString();

        makeExplosion(tempEnemy.x, tempEnemy.y);

        gameState = STATE_REMOVE_PLAYER;

    }

}

}
```

## 创建爆炸效果

最后的这一点代码就是 makeExplosions ( ) 函数了，这个函数需要传入 x 和 y 两个参数以便创建一个爆炸效果影片。它的作用还包括通过 play ( ) 方法在爆炸影片播放的时候同时播放一个爆炸音效。这看起来就跟创建一个本游戏中其它显示对象的方法一样，主要的不同在于，你不用给它加入一个速度值，因为爆炸效果不需要移动。但是，这个爆炸效果是有“生命”的（回忆一下它有七个帧）。这个影片剪辑在被创建的时候会自动的播放，所以我们不得不把它创建之后放入到一个 explosions 数组中以便之后控制它。



```
public function makeExplosion(ex:Number, ey:Number) {  
  
    var tempExplosion:MovieClip;  
  
    tempExplosion = new ExplosionImage();  
  
    tempExplosion.x = ex;  
  
    tempExplosion.y = ey;  
  
    addChild(tempExplosion);  
  
    explosions.push(tempExplosion);  
  
    var sound:Sound = new Explode();  
  
    sound.play();  
  
}
```

这又有一点跟扎气球游戏不一样的地方了，但是你应该能明白是怎么回事了。我们一直在试图给你举例说明一点：你可以通过一个完整游戏的代码马上开发出另一个游戏来。我们从点击游戏的代码开始，通过增加了一些方法变成了扎气球游戏，然后又添加了一些方法把它改成了像素射手游戏。通过用这种方法，你可以很容易地把像素射手变成更复杂的游戏。

## 总结

本小节的基本内容就是告诉你如何着手 AS3 游戏制作。后面会延伸这些游戏概念并制作不同类型的游戏，使你可以学的更多，前提别把这本书扔掉去看别的。然而，本节中的简单游戏中，很多地方都是需要改进的。如果你希望你的游戏更好，下面的话能提示你：

不是只有一种效果去重置游戏或者升级。

给计分板加入 text 是很笨的办法。

播放声音的方式虽然简洁但是很低效，因为你是用以一次创建一次。

如果不用 Flash IDE 开发使用影片剪辑的游戏会有点难度，许多人都是用 FlashIDE 和其他工具结合来做的更好的。

游戏的性能很差劲，如果你一直运行下去就会变得又慢又卡。

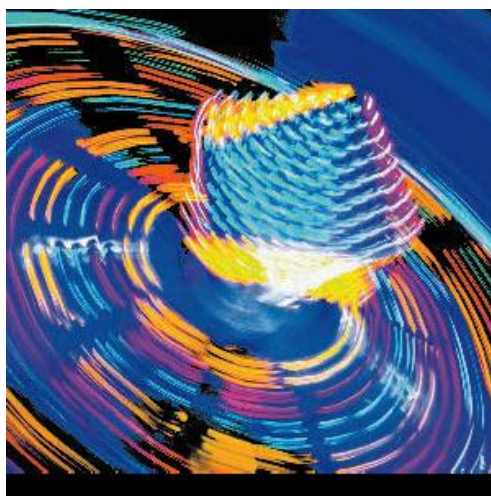


矩形碰撞检测对于不规则的图形时工作的并不好。

本书的后面会专注于建立游戏框架和解决这个问题（还有其他的问题）。不过呢，现在你需要停下来想想这节都学到什么了，并尝试制作一些不同风格的游戏。

现在嘛，你已经使用 AS3 游戏框架制作了你的第一个游戏，这还只是第一节！你就已经学会了游戏结构的基本内容（状态循环，游戏频率，事件模型）。也了解了一点哲学观点（第二游戏说）。就像你看到的，我们虽然讲的很快，但是可以跟上我们，因为这是很有效果的方式。

让我们诚实的反思一下；这节里的代码展示的核心概念不是用的最佳方式来表现的。因此在之后的章节中，我们要建立一个更优秀的游戏框架和更好玩的游戏来改善这一点。但是你要保证这个学习过程你是百分之百的独立完成，运行并修改的。现在，如果你准备好了，请跟我来，让我们去制作更多的游戏吧！



## 第二章

# 创建一个 AS3 游戏框架

---

在之前的章节中，我们已经建立了一个足够框架来制作游戏。然而，我们的任务还不止这些。你可能买了大量的书籍，并且受够了里面的教科书式的讲解。但是呢，你买了这本书，因为你想做的更好。我们不会让你落后的。这个章节就是承接上一章节的内容并把它们变成一个可以再次利用的框架来制作游戏。



我们会在后面的章节中反复的使用这个框架。

这个游戏框架是制作第二游戏的基础。它是从第一节中我们强调了结构和重用性的那个框架中扩展而来的版本。既然这本书不叫什么完全面向对象设计模式或者游戏标准之类的名字，那么你就会看到很少的理论教条和大量的练习。这里面的许多技术已经在ActionScript程序上久经考验了。我们还跟AS1，AS2 战斗过，但是现在是AS3 了，所以你不必重蹈覆辙了。

我们的框架借鉴了一些设计模式，你能感觉到游戏是一个非常有表现力的玩意。我们希望这本书的读者，不要多想那些乱七八糟的条条框框。设计模式让应用程序的开发变得容易起来，但是我们不想让你迷迷糊糊的跟着他们屁股后面制作你的游戏。我们的框架教你理解一个游戏的完整结构和流程，绝不是仅仅告诉你，怎么打代码。

## 探索游戏框架

游戏框架是一个被封装好的包结构。一个包结构的最基本的特点就就是一系列条理分明的文件夹，里面包含着可重用的类文件。框架所需要的所有标准类文件都在这个包结构中。随着本书的进行我们要随时更新和继承这个结构。让我们来告诉你我们需要在这个框架中创建哪些类。

## GameFrameWork.as类

GameFrameWork.as类是框架的基础类。它包含了一个简单的状态机制和一个基本的即时游戏循环。我们的状态机制是一个简单的结构，它仅仅是设置状态常量来转换当前框架运行状态的。状态被一个状态函数控制着，这个函数定义在GameFrameWork.as文件中。我们会雇一个变量来引用当前进行的状态。因游戏频率作用，下次在调用游戏循环的时候就会根据当前的状态调用对应的函数方法。我们要建立一些状态比如title screen，instruction screen，game play，game over 等等。



当我们创建一个新游戏的时候我们的GameFrameWork文档类就会被Main.as文件调用。Main.as有一个init ( initialization的简写 ) 函数会初始化框架需要的东西。这个Main.as类负责在游戏与框架类之间进行消息的传递转发。如果你还糊涂着，别着急；我们马上就给你详细的解说。

## FrameWorkStates.as类

FrameWorkStates.as类中简单的定义一些整形值来表示状态机制中的状态。当我们第一次创建框架的时候我们会有 10 种不同的状态常量。所有的状态声明前面都是STATE\_SYSTEM。比如，我们有一个状态来显示标题菜单。这个状态常量就会叫做STATE\_SYSTEM\_TITLE。随着这节的进行我们还会加入更多的状态。

## BasicScreen类和SimpleBlitButton辅助类

BasicScreen类是用来在屏幕上显示状态画面的类。这些画面可能有一个基本的背景色和透明度 ( 如果需要的话 )，也有一些文本在画面的中央。每一个画面都有一个可以点击的按钮来让状态转换。这个类简单的不得了。不过它是有实用目的的。

SimpleBlitButton辅助类是用来创建一个可点击的按钮并能让其可以经过和按下。这个按钮使用纯代码创建的，并且在鼠标经过和按下的时候只有一个简单的背景颜色。我们做的这么寒酸是因为要展示一些平面图形技术 ( 更多精彩在后面 )，使用了BitmapData来盖面按钮事件发生时颜色的改变。

## ScoreBoard类和SideBySideScoreElement辅助类

ScoreBoard类在整个玩游戏的过程中给用户显示相关信息。使用这个类可以显示一些数据比如当前的游戏分数，游戏时间，和其它有用的信息，这样能给用户一个直观的感觉。虽然它是可用的，但是当



你制作你的商业性游戏的时候，你可能需要继承这个类并扩展更多的方法。这个在框架里的基本类是用来告诉你如何使用一个游戏中控制更新计分板。

SideBySideScoreElement辅助类用来显示并排列文字标签的。比如，能用来在计分板上显示一个单词“Score”，后面还要跟着显示玩家当前的分数。

## Game类

Game类是一个“外壳”式的类，游戏中所有的类都要继承它。它包含了一些基本的变量和函数来为GameFrameWork类服务。

## 自定义事件类

框架内的各个类之间是用事件来进行通讯的。Game类的实例会通过事件和ScoreBoard及 Main这两个类进行交互。我们将为每个游戏都建立一个Main.as类.这个类将会是GameFrameWork类的子类.

另外我们还将要用到一些简单的事件类，它们是Flash提供的标准事件类，可用作于基本的交互。之所以这么打算，是因为我们现在还不需要在事件里传递数据。比如Game类里会定义一个名为GAME\_OVER的常量，我们实例化一个标准的flash事件后，将它的name属性设为GAME\_OVER，再通过标准的dispatchEvent方法把这事件发送出去。这里说的这个GAME\_OVER事件将会在当前游戏结束时用来告知Main类在状态循环逻辑里将当前游戏状态切换到游戏结束(STATE\_SYSTEM\_GAME\_OVER)。

我们也要再自定义三个不同的事件作为框架内容的一部分。它们将要被Game类实例用来向Main(或GameFrameWork)类实例发送数据，Main(或GameFrameWork)实例会根据收到的数据来进行下一步动作，如果有必要，也会把这些数据再传递给框架内的其它类。这样一来，我们就把Main.as当成了消息中转站。



## CustomEventButtononId.as类

这个自定义事件能够传递一个用来识别身份的整数值给监听器，通常用在多个按钮共用同一个监听器的情况下。它会在框架内被用到，特别是在GameFrameWork类、Main类里边，有了它就可以允许用到SimpleBlitButton类的多个BasicScreen实例之间共享同一个事件监听器。看到GameFrameWork类的代码你就会明白了。另外，你在其它的游戏或应用中需要在事件里发送一个简单的整数ID的时候也可以使用它。

## CustomEventLevelScreenUpdate.as类

这个自定义事件类通过Game.as类实例给BasicScreen实例发送数据。这个BasicScreen类的实例我们命名为levelInScreen。这个实例能够在每个游戏等级中显示自定义的文本。

这个事件的触发方法会通过一个发送一个数值来改变这个文本内容。Main.as类（GameFrameWork.as类的子类）会给这个监听这个时间并给levelInScreen传递数据。

## CustomEventScoreBoardUpdate.as类

这个自定义事件会通过Game.as类实例来更新ScoreBoard类实例的值。

Main.as类会监听这个事件并给ScoreBoard类的实例传递数据。

## 框架包结构

在我们创建一个包结构的时候要组织一下所有的代码。可重用的框架类会放在一个包中，游戏需要创建的指定类我们要放在单独的包中。

让我们马上去创建这两个包结构吧。

## 源文件夹



在你的某个盘符下选择一个路径，然后建立一个叫source的文件夹。这个文件夹以后就是本书中所有代码的老窝了。

## classes包

可重用的类要放在一个叫classes的包中。在source文件夹中再建立一个文件夹命名为classes。你现在应该有一个看起来像这样的文件夹结构：

```
[source]
```

```
    [classes]
```

然后我们要建立一个实际的包来包含所有可重用框架类的源代码。我们要把这个包命名为com.efg.framework。

那么怎么建立呢？首先你要在classes文件夹中建立一个叫com的文件夹，然后再在com文件夹中建立一个叫efg的文件夹。最后嘛，再在efg文件夹中建立一个叫framework的文件夹就好了，顺便说一下，“efg”是本书标题中“Essential Flash Games”的缩写。现在你的文件目录结构看起来应该是这样的：

```
[source]
```

```
    [classes]
```

```
        [com]
```

```
            [efg]
```

```
                [framework]
```

当我们开始创建这些框架必要的类文件的时候，他们就会放在这个framework文件夹中。你会看到当我们创建这些类实例的时候，包名是这样写的：

```
package com.efg.framework {
```



## projects包

第二个包我们要叫它“projects”。现在你可以自己在根目录( source文件夹 )中建立这个叫project的文件夹了。projects包结构不用像classes包那样是连续的。

这种组织方式来源于一种游戏开发习惯。在projects文件夹中，我们要给本书中不同游戏分别建立一个不同的文件夹。这个第一个游戏我们要命名为stubgame。

stub通常用来形容一个函数或者一个类中包含了很少（有也不多）的代码，地位仅仅比占位符略高。我们这个游戏当然要比一个占位符好一点不过也好不到哪去。它只是用来验证一个框架的基本运作方式。好了废话少说让我们继续吧，在projects文件夹中再创建一个叫stubgame的文件夹。现在你的目录结构就应该是这样的了：

[source]

[classes]

...

[projects]

[stubgame]

然后，我们要建立两个文件夹，每个文件夹存放着我们的游戏的不同版本，为什么要这么做呢？本书提倡用多种开发工具来开发Flash游戏的。这些工具的开发流程和开发环境（IDEs）有很多方法都是共同的。

在这本书中我们主要使用两种工具：Flash IDE(这个IDE带有库，时间轴，绘图工具和其他的小工具)和Flash Develop（一个非常流行的，免费的，专业编写ActionScript代码的开发工具）。

你可以使用任意的开发工具，但是你可能需要按照这个开发工具的文档来设置工程参数。你要仔细



检查你的游戏项目的包结构，因为你使用的代码编辑环境与代码联系的非常紧密，不能有疏漏。

建立游戏的包链接虽然是很简单的，但是在不同的IDE中会有一点差别。杰夫大多数情况下用Flash Develop 和Adobe提供的免费Flex SDK来开发Flash游戏，而史蒂芬，却只用Flash IDE来完成这种工作。

(译者注：这个例子举的真囧)

我们已经整合了所有章节中给你的代码，他们都可以用Flex SDK 和 Flash IDE来运行。因此，下面你要在stubgame文件夹中建立的两个文件夹一个要命名为flashIDE，另一个则是flexSDK。你不必两个都弄好，你只需根据你使用的开发工具来建立文件夹就好，因为两种方式是有区别的，所以你只要把注意力放在你最常用的工具上面。

你现在的工程文件夹结构应该是这样的：

[projects]

[stubgame]

[flashIDE]

[flexSDK]

## Flash IDE 包结构

Flash IDE 的包结构要在flashIDE文件夹中正确的展开。包名的最后一个字段是非常简单的。包的导入结构将会是 com.efg.games.[游戏名称 ( game name ) ]。

比如说吧，本章我们要建立的stub游戏，报名就要写成 com.efg.games.stubgame。现在继续建立这些文件夹吧。完成的时候你的包的文件夹结构是这样的：

[projects]

[stubgame]



[flashIDE]

[com]

[efg]

[games]

[stubgame]

[flexSDK]

## Flex SDK 包结构

Flex SDK包结构跟Flash IDE 包结构不同之处就是简单明了。Flash Develop 和其他Flex 工具可以特别设置文件夹来组织工程的结构。不管如何，在我们的游戏中这些文件夹都有一个通用的结构，我们要给Flash Develop 的项目中加入flexSDK 文件夹里的src文件夹。

你不用手动建立src文件夹或者其他的包结构，Flash Develop会自动的创建它们当你new了一个新工程的时候。我们会在之后的部分中详细说明，这个部分叫做“在Flash Develop中设置游戏开发”。现在嘛，我们只说这么一种查看方法（包括Flash Develop中其他的文件夹比如 bin ， obj ， 和 lib ）。如果你用Flash Develop建立一个新的Flex SDK工程，你会看到以下的文件结构：

[projects]

[stubgame]

[flexSDK]

[bin]

[obj]



[lib]

[src]

[com]

[efg]

[games]

[stubgame]

现在回到我们已经创建的好的 Flash IDE 下的包结构。我们游戏的包名将写作 com.efg.games.stubgame。

无论是在Flash IDE还是在Flex SDK中导入的包名都是一样的：

```
package com.efg.games.stubgame  
  
{
```

## Main.as和StubGame.as文件

首先我们要给subgame包的框架类中添加两个我们游戏中需要的亚类（或称子类）。Main.as类文档作为GameFrameWork.as框架类的子类被创建。而StubGame.as类是Game.as框架类的子类。

（译者注：这里作者写的很罗嗦，每次都要先说一遍亚类（subclass）再注明一句或子类（or child）

校对酌情删减吧）。

## 使用框架包新建一个工程

你已经了解框架复用类和创建工程所需的基本包结构知识了。让我们来给stub游戏的工程运作起来



吧。stub game就是第一章那个简单到弱智的鼠标点击 10 次完事的小游戏。

## 在Flash IDE中建立stub游戏

那么，怎么做呢？跟我一步步来：

1. 打开你的Flash 软件。我用的是CS3 版本，这个无所谓CS4 还是CS5 都是一样的。
2. 建立一个fla文件，并把它存放在/source/projects/stubgame/flashIDE/文件夹中，命名为stubgame。
3. 在/source/projects/stubgame/flashIDE/文件夹中，紧跟着给你的游戏建立一个包结构：  
/com/efg/games/stubgame/  
4. 把Flash的帧频设置为 30FPS。舞台的宽与高都设置成 400 像素。
5. 把文档类设置为com.efg.games.stubgame.Main
6. 因为我们还没创建Main.as类文件所以你会看见一个警告信息。我们会在章节的后面进行创建。
7. 把框架中的复用类的路径添加给fla文件
  - 在发布设置中，选择【Flash】>【ActionScript 3 设置】。
  - 点击浏览到路径按钮找到之前我们在包结构中创建的/source文件夹。选择classes文件夹，然后单击选择按钮。

现在com.efg.framework包就能够在我们的创建游戏中被使用了。尽管我们还没有创建具体的框架类文件，不过我们很快就完成这项工作。

## 在Flash Develop中建立stub游戏工程

下面在Flash Develop中建立跟上面同样工程的步骤：



1. 在[source][projects][stubgame]文件夹里再建立一个名为[flexSDK]的文件夹（如果你之前没建立过）。

2. 打开Flash Develop，然后建立一个新工程：

- 选择Flex 3 工程
- 给工程命名为stubgame。
- 工程放置路径应该是/source/projects/stubgame/flexSDK文件夹。
- 包应该为com.efg.games.stubgame。
- 如果不想让Flash Develop 自动创建工程文件夹结构，请确保“给工程创建文件夹”的复选框是未被勾选的。
- 单击完成按钮建立工程。

3. 把框架的类路径添加给工程：

- 选择【工程】>【属性】>【类路径】菜单项。
- 点击添加类路径按钮。
- 找到之前创建的/source文件夹，然后选择classes 子文件夹。
- 点击完成按钮然后再点击应用按钮。

译者注：这段我之所以是一小步翻译一点是考虑到Flash Develop有中英文两版，要不要把按钮菜单的名称翻译过来还拿不准。

现在你已经能在建立工程的时候运用到基础框架了。然后我们要就框架类的结构与代码编写来稍稍

下面是两点提示：

无论是在Flex Builder，Flash Builder 或者其他的IDE中，请事先阅读一下工具本身的使用手册，以便与你能在使用其他开发工具的时候也能按照我们游戏的需要来进行工程的设置。

一个开发Flash的普遍方法是，使用Flash IDE来管理资源库，用Flash Develop 来进行代码的编写。如果这也是你的工作方式，你就要使用Flash IDE的文件夹和包结构而不是Flex SDK的文件夹结构。



的讨论一下。

## 建立游戏循环 ( game timers )

当我们需要建立一个基于帧的游戏循环的时候，大多数Flash 开发者会使用两种方法。“基于帧”的方式就是说我们把一个时间轴分割成一个个独立的部分，通过不断的播放这些琐碎的时间段来进行游戏逻辑的循环。

还有其他的方法来进行游戏的实时更新，但是本书中我们会常用到时间切片或者说基于帧的循环方法。这种基本的循环方法会试图在每一个时间片段（或称作帧）里进行游戏的流程和屏幕的刷新。我们在第 11 章里还会探索一个步时循环（time-step）和基于休眠的循环（sleep-based）。

第一个循环方法就是Event.ENTER\_FRAME 事件循环了，这种标准的Event.ENTER\_FRAME事件句柄会根据swf文件设置的帧频来进行游戏的循环。这种简便的循环方式已经久经考验了。第二种循环方法是用Timer类来实现的，Timer类循环的速度是通过毫秒为单位的延时参数来控制的。

比如，如果把毫秒的延时参数设为 100，那么这个计时器就会在一秒钟内执行十次（1000 毫秒是 1 秒）。我们的框架会使用到 Timer 的循环实例的。我们会使用一个叫做 TimerEvent.TIMERUpdateAfterEvent的函数。顾名思义，这个函数的作用就是让屏幕的刷新趋于平滑。

## 定义“帧循环次数”（“frame timer tick”）

你会看到诸如“帧时间限制”，“时间限制”和“帧限制”的词语在本书中被使用。当我们说“限制”或者“帧限制”的时候，我们的意思就是指一帧的程序运行的时间量。比如说当我们把游戏的帧频设置为 30 帧每秒的时候，那么在每次循环的时候只有 33.33 毫秒（或者是 1000/30 毫秒）的时间来运行我们的程序。



译者注：这段我纠结啊.....ticks到底怎么翻译好呢。

## 状态机制

一个传统的状态机只是基本的状态控制机制而已，或者说是当前行为控制系统。有时候这也被叫做有限状态机。有限状态机还可以被用于组建复杂的数学计算，而且更接近于人工智能范畴。

有限的意思是指，这个系统在任何情况下只能执行一种状态。我们的游戏框架就是被建立在一个简单的状态机上，通过独立的函数或者方法来执行每种状态行为。

还有许多其他的状态机；一些是用整个类来作为一个独立的状态（有时被叫做面向对象的状态机），一些使用简单的switch：case分支选择来控制状态。而我们会使用一个借鉴了这两种方法的第三种办法。

我们的状态机每次只调用一个函数。不像面向对象状态机那样，我们的每个状态在其单框架类中都独立的方法或函数。每一个状态方法都能控制框架中的某些元素比如说BasicScreen 类和Game类的实例。我们要用switch：case语句来进行状态之间的转换。不像之前提及的那种非常简单的switch：case状态机，要在每个帧执行时调用switch/state控制结构，我们只需要在状态切换的时候再调用它就可以了。

switch：case会在每帧执行被调用的时候选择要不要切换游戏状态。

GameFrameWork.as 文件里包含了一个控制整个游戏流程的状态机。我们会把它放到FrameWorkStates.as文件中并让它可以切换我们的游戏状态。

每个单独的游戏包我们都会建立一个Main.as文件（在这个游戏自己的包结构中），这个文件继承了GameFrameWork.as。我们也要给每个游戏创建一个Game.as子类。我们创建的Game子类可以根据需要来声明采用它特有的内部状态。

## FrameWorkStates.as类文件

这个类文件是一个给游戏框架中声明的常量的集合。它们是被GameFrameWork.as类文件所使用的东西。下面的代码列出了这个文件的所有代码；你要是想写一个的话，就要按照之前说明的包结构来创建这个文件。

```
package com.efg.framework

{

    /**
     * ...
     * @author Jeff and Steve Fulton
     *
     */

    public class FrameWorkStates {

        public static const STATE_SYSTEM_WAIT_FOR_CLOSE:int = 0;

        public static const STATE_SYSTEM_TITLE:int = 1;

        public static const STATE_SYSTEM_INSTRUCTIONS:int = 2;

        public static const STATE_SYSTEM_NEW_GAME:int = 3;

        public static const STATE_SYSTEM_GAME_OVER:int = 4;

        public static const STATE_SYSTEM_NEW_LEVEL:int = 5;

        public static const STATE_SYSTEM_LEVEL_IN:int = 6;

        public static const STATE_SYSTEM_GAME_PLAY:int = 7;

        public static const STATE_SYSTEM_LEVEL_OUT:int = 8;
```



```
public static const STATE_SYSTEM_WAIT:int = 9;
```

```
}
```

```
}
```

首先你要注意的就是第一行的包名。它的规则是先前创建的按照文件目录的结构来的。不管你用Flash , Flex , Flash Builder , Flash Develop , TextMate甚至是记事本中的哪一个工具, 包名都是一样的。包名不依赖代码开发环境但是我们这么做是为了更好的组织代码。按照我们先前创建的路径保存这个文件:

```
/source/classes/com/efg/framework/FrameWorkStates.as
```

## 状态变量

状态变量都是被游戏的状态机循环切换状态时用到的一些常量。我们在这个示例文件中声明了一些最基本的状态, 但是你可以声明更多你需要的状态。随着本章节的进行, 将需要更多的状态加入。

STATE\_SYSTEM\_TITLE: 这个状态被用来显示一个基本的标题画面, 还有一个OK按钮以使用户点击。一旦用户点击按钮, 标题画面就会结束, 状态就会进行切换。

STATE\_SYSTEM\_WAIT\_FOR\_CLOSE: 这个是等待任何BasicScreen类实例上OK按钮被点击前的状态。

STATE\_SYSTEM\_INSTRUCTIONS: 这个状态被用来显示跟SYSTEM\_TITLE状态下OK按钮一样的东西。它也会在用户点击OK按钮后切换到STATE\_SYSTEM\_WAIT\_FOR\_CLOSE状态。

STATE\_SYSTEM\_NEW\_GAME: 这个状态会调用游戏的逻辑类并运行, 其game.newGame()函数。而且在之后立刻切换到 NEW\_LEVEL 状态。



STATE\_SYSTEM\_NEW\_LEVEL:在这个状态下，我们可以调用game.newLevel()函数来给游戏设置一个新的等级。

STATE\_SYSTEM\_LEVEL\_IN:这个状态会在必要的情况下显示一些基本的等级信息。在这个基础的游戏，我们简单的显示等级画面然后几秒之后就让其移除。等待的过程是在STATE\_SYSTEM\_WAIT状态下通过帧频调用和一个具体的数值来完成的。

STATE\_SYSTEM\_GAME\_PLAY:这个状态简单的调用游戏逻辑里的runGame函数好让游戏开始其自己的逻辑与状态。

STATE\_SYSTEM\_GAME\_OVER:游戏结束的状态会显示一个游戏结束的画面然后等待用户点击OK按钮来回到初始画面。它会很快切换到STATE\_SYSTEM\_WAIT\_FOR\_CLOSE直到OK按钮被点击了。

STATE\_SYSTEM\_WAIT:这个状态通过一个叫WAIT\_COMPLETE的自定义的事件实例和一个数值来实现等待过程。

## GameFrameWork.as类文件

GameFrameWork.as是我们游戏类文档的父类。我们游戏的Main.as（游戏的类文档）类就是继承了这个类而来，然后再根据每个游戏的不同而进行必要的修改。后面列出了全部的代码。我们会详细的讲述它。这个类文件的路径是

```
/source/classes/com/efg/framework/GameFrameWork.as
```

GameFrameWork.as类也是我们游戏的类文档Main.as的父类，它在后面被列出来了。在后面的章节中，我们会给这个文件加入一些函数甚至是创建一个完全不同的timer。

Main.as类通过extends语法成为了这个类的子类，并且重载了其父类中空白的init函数（译者注：在编程中，init函数通常表示初始化函数）。而且这个类还包含了FrameWorkStates类中所有的状态变量。



GameFrameWork.as类里的所有函数都写成了public模式，以便我们在必要的时候可以在Main.as类中重载它们。这样以来，我们就可以自定义一些我们需要的状态行为函数。

比如说，在后面的章节，我们想在标题画面中播放音乐。而这个方法会被加入到systemTitle状态函数中。因为不是所有的游戏都需要这个方法，所以我们不需要把它添加到GameFrameWork.as文件的systemTitleFunction中。而是在Main.as类中通过重载GameFrameWork.as类中的方法写一个新的systemTitleFunction函数来。这个新的函数就可以通过在GameFrameWork.as中使用super.systemTitle()方法来播放声音。

```
package com.efg.framework  
{  
  
    import flash.display.Bitmap;  
    import flash.display.BitmapData;  
    import flash.display.MovieClip;  
    import flash.events.Event;  
    import flash.geom.Point;  
    import flash.text.TextFormat;  
    import flash.utils.Timer;  
    import flash.events.TimerEvent;  
  
    public class GameFrameWork extends MovieClip {  
  
        public static const EVENT_WAIT_COMPLETE:String = "wait complete";
```



```
public var systemFunction:Function;

public var currentSystemState:int;

public var nextSystemState:int;

public var lastSystemState:int;

public var appBackBitmapData:BitmapData;

public var appBackBitmap:Bitmap; ;

public var frameRate:int;

public var timerPeriod:Number;

public var gameTimer:Timer;

public var titleScreen:BasicScreen;

public var gameOverScreen:BasicScreen;

public var instructionsScreen:BasicScreen;

public var levelInScreen:BasicScreen;

public var screenTextFormat:TextFormat;

public var screenButtonFormat:TextFormat;

public var levelInText:String;

public var scoreBoard:ScoreBoard;

public var scoreBoardTextFormat:TextFormat;

//Game is our custom class to hold all logic for the game.

public var game:Game;

//waitTime is used in conjunction with the
```



```
//STATE_SYSTEM_WAIT state
```

```
// it suspends the game and allows animation or other
```

```
//processing to finish
```

```
public var waitTime:int;
```

```
public var waitCount:int=0;
```

```
public function GameFrameWork() {
```

```
}
```

```
public function init():void {
```

```
    //stub to override
```

```
}
```

```
public function
```

```
setApplicationBackGround(width:Number,height:Number,isTransparent:Boolean = false,
```

```
color:uint = 0x000000):void {
```

```
    appBackBitmapData = new BitmapData(width, height, isTransparent, color);
```

```
    appBackBitmap = new Bitmap(appBackBitmapData);
```

```
    addChild(appBackBitmap);
```

```
}
```

```
public function startTimer():void {
```



```
timerPeriod = 1000 / frameRate;

gameTimer=new Timer(timerPeriod);

gameTimer.addEventListener(TimerEvent.TIMER, runGame);

gameTimer.start();
}

public function runGame(e:TimerEvent):void {

    systemFunction();

    e.updateAfterEvent();
}

//switchSystem state is called only when the
//state is to be changed

//(not every frame like in some switch/case
//based simple state machines

public function switchSystemState(stateval:int):void {

    lastSystemState = currentSystemState;

    currentSystemState = stateval;

    switch(stateval) {

        case FrameWorkStates.STATE_SYSTEM_WAIT:

            systemFunction = systemWait;

            break;

        case FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE:
```



```
systemFunction = systemWaitForClose;
```

```
break;
```

```
case FrameWorkStates.STATE_SYSTEM_TITLE:
```

```
systemFunction = systemTitle;
```

```
break;
```

```
case FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS:
```

```
systemFunction = systemInstructions;
```

```
break;
```

```
case FrameWorkStates.STATE_SYSTEM_NEW_GAME:
```

```
systemFunction = systemNewGame;
```

```
break;
```

```
case FrameWorkStates.STATE_SYSTEM_NEW_LEVEL:
```

```
systemFunction = systemNewLevel;
```

```
break;
```

```
case FrameWorkStates.STATE_SYSTEM_LEVEL_IN:
```

```
systemFunction = systemLevelIn;
```

```
break;
```

```
case FrameWorkStates.STATE_SYSTEM_GAME_PLAY:
```

```
systemFunction = systemGamePlay;
```

```
break
```

```
case FrameWorkStates.STATE_SYSTEM_GAME_OVER:
```



```
        systemFunction = systemGameOver;

        break;
    }
}

public function systemTitle():void {

    addChild(titleScreen);

    titleScreen.addEventListener(CustomEventButtonId.BUTTON_ID,
        okButtonClickListener, false, 0, true);

    switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);

    nextSystemState = FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS;
}

public function systemInstructions():void {

    addChild(instructionsScreen);

    instructionsScreen.addEventListener(CustomEventButtonId.BUTTON_ID, okButtonClickListener, false, 0, true);

    switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);

    nextSystemState = FrameWorkStates.STATE_SYSTEM_NEW_GAME;
}

public function systemNewGame():void {

    addChild(game);
```



```
game.addEventListener(CustomEventScoreBoardUpdate.□  
UPDATE_TEXT,scoreBoardUpdateListener,false, 0, true);  
game.addEventListener(CustomEventLevelScreenUpdate.□  
UPDATE_TEXT,levelScreenUpdateListener,false, 0, true);  
game.addEventListener(Game.GAME_OVER, gameOverListener,false, 0, true);  
game.addEventListener(Game.NEW_LEVEL, newLevelListener,false, 0, true);  
game.newGame();  
switchSystemState(FrameWorkStates.STATE_SYSTEM_NEW_LEVEL);  
}  
public function systemNewLevel():void {  
    game.newLevel();  
    switchSystemState(FrameWorkStates.STATE_SYSTEM_LEVEL_IN);  
}  
public function systemLevelIn():void {  
    addChild(levelInScreen);  
    waitTime = 30;  
    switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT);  
    nextSystemState = FrameWorkStates.STATE_SYSTEM_GAME_PLAY;  
    addEventListener(EVENT_WAIT_COMPLETE, □  
    waitCompleteListener, false, 0, true);  
}
```



```
public function systemGameOver():void {  
  
    removeChild(game);  
  
    addChild(gameOverScreen);  
  
    gameOverScreen.addEventListener(CustomEventButtonId.BUTTON_ID,  
    onClickButtonClickListener, false, 0, true);  
  
    switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);  
  
    nextSystemState = FrameWorkStates.STATE_SYSTEM_TITLE;  
  
}  
  
public function systemGamePlay():void {  
  
    game.runGame();  
  
}  
  
public function systemWaitForClose():void {  
  
    //do nothing  
  
}  
  
public function systemWait():void {  
  
    waitCount++;  
  
    if (waitCount > waitTime) {  
  
        waitCount = 0;  
  
        dispatchEvent(new Event(EVENT_WAIT_COMPLETE));  
  
    }  
  
}
```



```
public function okButtonClickListener(e:CustomEventButtonId):void {  
  
    switch(e.id) {  
  
        case FrameWorkStates.STATE_SYSTEM_TITLE:  
  
            removeChild(titleScreen);  
  
            titleScreen.removeEventListener(CustomEventButtonId.BUTTON_ID,okButtonClickListener);  
  
            break;  
  
        case FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS:  
  
            removeChild(instructionsScreen);  
  
            instructionsScreen.removeEventListener(CustomEventButtonId.BUTTON_ID,okButtonClickListener);  
  
            break;  
  
        case FrameWorkStates.STATE_SYSTEM_GAME_OVER:  
  
            removeChild(gameOverScreen);  
  
            gameOverScreen.removeEventListener(CustomEventButtonId.BUTTON_ID,okButtonClickListener);  
  
            break;  
  
    }  
  
    switchSystemState(nextSystemState);  
  
}  
  
public function scoreBoardUpdateListener(e:CustomEventScoreBoardUpdate):void {
```



```
scoreBoard.update(e.element, e.value);
}

public function levelScreenUpdateListener(e:CustomEventLevelScreenUpdate):void {

    levelInScreen.setText(levelInText + e.text);
}

//gameOverListener listens for Game.GAMEOVER simple
//custom events calls and changes state accordingly
public function gameOverListener(e:Event):void {

    switchSystemState(FrameWorkStates.STATE_SYSTEM_GAME_OVER);

    game.removeEventListener(CustomEventScoreBoardUpdate.□
    UPDATE_TEXT,scoreBoardUpdateListener);

    game.removeEventListener(CustomEventLevelScreenUpdate.□
    UPDATE_TEXT, levelScreenUpdateListener);

    game.removeEventListener(Game.GAME_OVER, gameOverListener);

    game.removeEventListener(Game.NEW_LEVEL, newLevelListener);
}

//newLevelListener listens for Game.NEWLEVEL
//simple custom events calls and changes state accordingly
public function newLevelListener(e:Event):void {

    switchSystemState(FrameWorkStates.STATE_SYSTEM_NEW_LEVEL);
```



```
}  
  
public function waitCompleteListener(e:Event):void {  
  
    switch(lastSystemState) {  
  
        case FrameWorkStates.STATE_SYSTEM_LEVEL_IN:  
  
            removeChild(levelInScreen);  
  
            break  
  
        }  
  
        removeEventListener(EVENT_WAIT_COMPLETE,waitCompleteListener);  
  
        switchSystemState(nextSystemState);  
  
    }  
  
}  
  
}
```

## 类包的导入

类包的导入部分包括Main类必要的Flash核心类。

注意前面的包名，它的名称跟我们在章节前面建立的框架包结构是一致的：

```
package com.efg.framework
```

```
{
```

我们还必须导入所有框架运行时需要的类。你马上就会看见。



## 变量的定义

变量定义的部分定义了类中所有的全局变量。包括状态机必要的状态变量，画面显示用的变量还有游戏循环中所用的变量。

我们利用常量来定义当前的状态并声明一个变量来存储状态信息。这些状态都在之前的部分被创建的FrameWorkStates.as文件中被声明了。你可以加入更多的状态，但是我们声明的这些足以应付本书中的许多游戏了。

这里有两个特别的状态是用来做系统事件触发等待或者动画完成等待用的，它们分别是STATE\_SYSTEM\_WAIT\_FOR\_CLOSE和STATE\_SYSTEM\_WAIT。我们一般在游戏循环中当前状态下调用一个常规的函数去调用systemFunction方法。

通过用一个整形变量来保存当前状态（currentSystemState）的这种方式，之前的状态（lastSystemState）和下一个状态（nextSystemState）这种方式来控制游戏流程。这些状态不会因为游戏的暂停而混乱。这个我们会在第 11 章讲解用不同的方式给框架加入状态。

如果你在使用Flash IDE并且库中有很多在第一帧就要被输出的资源，那么你的类文档就必须继承MovieClip而不是Sprite，即使你不打算使用时间线。我们已经给GameFrameWork类继承了MovieClip所以它在Flex SDK和Flash IDE中都好使。

## 状态控制变量

控制变量会让系统函数始终包含一个当前状态。主要的控制变量就是

名为systemFunction的Function类实例。这个Function会在每帧执行的时候调用相应的状态行为函数。我们用switch:case语句来实现状态的转换与调用。当前的状态行为函数会被switchSystemState()函数控制并更新状态。



优化! `switchSystemState()` 是我们框架众多优化中的第一步。所有的优化都是为了让Flash游戏运行的更有效。这些游戏框架中的优化会让每一帧的运行效率更完美。

`currentSystemState` 整形变量保存了当前运行状态的常量值。`nextSystemState` 则是保存了下一个要切换的状态常量值。`lastSystemState` 则记录了上一个运行的状态常量值。这是为了让游戏循环能够回到先前的状态中，虽然这种情况很少见。

`lastSystemState` 变量在 `STATE_SYSTEM_WAIT` 之类的状态中被用到。`STATE_SYSTEM_LEVEL_IN` 状态会在进入 `STATE_SYSTEM_GAME_PLAY` 状态之前执行一个 30 毫秒的延时。同样的，`nextSystemState` 和 `lastSystemState` 也在 `STATE_SYSTEM_WAIT` 和 `lastSystemState` 中被用到。当这个 30 毫秒的延时到期后，`waitCompleteListener` 函数就会被调用。它会用 `lastSystemState` 变量去判断之前进行的是哪一个状态。我们会在后面的章节详细讲解 `waitCompleteListener` 函数。

## 背景填充变量

所有的Flash应用都有一个背景色。不管怎么样在我们要建立的游戏中，框架是可以控制背景色的。

你不应该依赖于HTML中的设置来给你的Flash应用作为背景色。HTML的代码是脱离你的应用控制的。如果你创建了一个图形游戏并发布在了一个游戏门户网上，你就不能控制游戏的背景色了，它会通过门户网上的HTML代码变成这个门户网设置的标准背景色。框架则通过一个有色位图的填充来让你覆盖HTML上的颜色。我们只要定义一个名为 `BitmapData` 的对象和一个名为 `Bitmap` 的对象来把位图信息添加到显示列表就可以了。

我们不会在 `GameFrameWork.as` 中定义背景，而是在 `GameFrameWork` 的子类 `Main.as` 中的 `init` 函数中定义我们需要的颜色。



## 频率变量

Timer会控制游戏的帧频并通过TimerEvent.updateAfterEvent方法来让图形的显示更加平滑。

最重要的是注意我们要使用内置的Timer类 ( gameTimer )。我们不用标准的EnterFrame事件。这样我们就可以建立自己的游戏帧频,而且可以自由的控制游戏循环的执行速率而不是依靠.swf文件的帧频 ( FPS )。

.swf文件可能设置帧频为 25 ( 举个例子而已 ), 但是游戏的循环可能会每秒执行 30 次。这样的话, 我们首先设置我们想要的帧频为每秒 30 次。当Main.as类调用startTimer函数 ( 当我们讲到init函数部分时会详细的说明 ) 时, gameTimer就会开始工作。首先, 我们要用 1000/帧频来计算出timerPeriod的值。

随着updateAfterEvent函数的调用 ( 在runGame函数中 ), 我们可以让Flash 的显示机制在每个时间周期内平滑的进行屏幕着色,而不是根据.swf文件设置的帧频来刷新屏幕。回到之前我们的例子,如果游戏SWF文件设置的帧频 ( FPS ) 为 25, 但是frameRate设置的是 30, 那么updateAfterEvent 就会让Flash的显示机制用 30 次每秒的速率来刷新屏幕而不是swf文件所设置的 25。

timerPeriod ( 时间周期 ) 会通过Timer实例来实现而且游戏也会企图用这个速率来运行。我们说“企图”是一位内如果游戏在画面上包含了太多的移动对象或者过于复杂的游戏逻辑, 那么就会出现明显的丢帧 ( 就是我们常说的“卡”) 现象。在后面的章节中, 我们会给runGame函数加入一个功能来缓解这一情况。

## 屏幕定义变量

屏幕定义变量建立了我们BasicScreen类的实例。这一个相当简单的类来放置一个文本框和一个ok



按钮在屏幕上。我们会使用这个简单的画面来作为标题，说明，等级和游戏结束的画面。我们会在游戏的init函数被重载是定义每个画面。注意levelInText是一个特别的变量。

给这个字符串设置一个默认值会让levelInScreen在每个新关卡时显示一段文字。这个文字会结合动态文本来建立一段文字。比如说在Level 1 中就显示单词“Level”作为默认文本再加上一个数字“1”作为动态文本。

```
public var titleScreen:BasicScreen;  
  
public var gameOverScreen:BasicScreen;  
  
public var instructionsScreen:BasicScreen;  
  
public var levelInScreen:BasicScreen;  
  
public var levelInText:String;  
  
public var screenTextFormat:TextFormat;  
  
public var screenButtonFormat:TextFormat;
```

我们还要建立两个TextFormat对象用来定义文本的字体格式和按钮上的字体格式。

## 计分板变量

scoreBorad是ScoreBoard类的实例，作为一个题头显示（HUD）让用户获得当前的分数信息。它是一个在每个游戏Main.as类中的init重载函数中定义的简单框架类。它根据游戏内容的变化而变化。我们还要定义一个TextFrmatObject来给计分板的文字定义文本格式：scoreBoardTextFormat。

## 游戏对象变量



游戏对象，只是一个因游戏所命名的变量，也是Game类的实例。Main.as类的init函数重载会实例化它。比如说，我们在本章节建立的StubGame.as就是Game.as类的一个子类。它会重载Game.as类的一些部分并定义其游戏的独立逻辑。

```
//Game is our custom class to hold all logic for the game.
```

```
private var game:Game;
```

## wait变量

这些变量是用来在STATE\_SYSTEM\_WAIT状态下进行等待作用的。waitTime可以在每次使用的时候进行设置。30 是其默认值。因为我们已经设置框架的帧频为每秒 30 帧，所以等待的时间就相当于 1 秒。如果帧频设置是 30 的话那么 30 帧运行的时间就等于 1 秒。waitCount变量在每帧执行后增加。当waitCount==waitTime的时候，控制器就会进入下一个状态。

```
//waitTime is used in conjunction with the STATE_SYSTEM_WAIT state
```

```
// it suspends the game and allows animation or other processing to
```

```
//finish
```

```
private var waitTime:int = 30;
```

```
private var waitCount:int=0;
```

## 构造函数定义

GameFrameWork.as的构造函数不包括任何代码。它只作为一个类似于占位的作用。我们会给每个游戏的GameFrameWork.as类都创建一个名为Main.as的子类。Main.as重载的构造函数中就会包含调用init函数的代码了。



```
public function GameFrameWork() {  
  
}
```

## init函数定义

Init ( ) 函数通过Main.as类重载GameFrameWork.as的方法而来。

## setApplicationBackGround ( 设置应用背景 ) 函数定义

这个函数利用参数类创建游戏的背景色。给背景定义宽度，高度，是否透明以及颜色值来实例化appBackBitmapData并将其加入显示列表。

## startTimer函数定义

这个函数通过Main.as的init函数来调用。它会根据frameRate变量来创建timerPeriod ( 时间周期 )。既然timerPeriod必须用毫秒表示 ( 1 秒的 1000 分之一等于timerPeriod的 1000 或者 1 秒 )，我们只要把frameRate用 1000 相除来算得timer的周期。假如我们的frameRate设置为 30，那么timerPeriod就是 33.33。

## runGame函数定义

runGame函数是状态机的核心。当systemFunction运行至switchSystemState ( 切换状态 ) 调用的时候，runGame函数就会在一个timerPeriod ( 大约是 33 毫秒 ) 内反复的调用它。

```
public function runGame(e:TimerEvent):void {
```



```
systemFunction();  
  
e.updateAfterEvent();  
  
}
```

`e.updateAfterEvent()`函数的调用会告诉Flash播放器在每帧执行完毕之后进行一次屏幕的更新，而不是等到下一帧才更新。系统的帧更新事件是基于SWF的舞台帧频而发生的。如果我们在这里不调用`updateAfterEvent`，屏幕就不会刷新直到下一帧开始运行。通过这种方式，我们让屏幕的刷新看起来更平滑。

## switchSystemState（选择系统状态）函数定义

如果`runGame`函数是`timer`的核心，那么`switchSystemState()`函数就是简化状态机的核心（译者注：这里与上一段的`runGame`函数讲解中有不一致）。它为状态传递一个常量值。通过使用这个值来选择`systemFunction`要做的事。

`switchSystemState`函数用来改变当前的`systemFunction`。一个状态常量传递进了这个函数，那么它就会改变`systemState`变量。也让`lastSystemState`和`nextSystemState`发生变化。作为参考，下面是`FrameWorkStates.as`类中定义的常量：

```
public static const STATE_SYSTEM_WAIT_FOR_CLOSE:int = 0;  
  
public static const STATE_SYSTEM_TITLE:int = 1;  
  
public static const STATE_SYSTEM_INSTRUCTIONS:int = 2;  
  
public static const STATE_SYSTEM_NEW_GAME:int = 3;  
  
public static const STATE_SYSTEM_GAME_OVER:int = 4;
```



```
public static const STATE_SYSTEM_NEW_LEVEL:int = 5;
```

```
public static const STATE_SYSTEM_LEVEL_IN:int = 6;
```

```
public static const STATE_SYSTEM_GAME_PLAY:int = 7;
```

```
public static const STATE_SYSTEM_LEVEL_OUT:int = 8;
```

```
public static const STATE_SYSTEM_WAIT:int = 9;
```

我们首先设置 `lastSystemState = currentSystemState`，这样我们可以在返回上个状态时有个参考。这种情况可能是我们在`STATE_SYSTEM_WAIT`状态的一个时间周期内想跳回我们在等待之前的状态。`systemLevelIn`函数就是个好例子。当我们调用`systemLevelIn`函数的时候，我们想在屏幕上移除`levelInScreen`之前等几毫秒。时间一到，`WAIT_COMPLETE`事件就会被触发。`waitCompleteListener`函数就需要知道先前的状态是什么以便程序往下运行。

然后我们设置`currentSystemState = stateval`。`stateval`是我们调用`switchSytemState`函数时传递的一个参数。这样的话`switch/case`可以把当前的`systemFunction`设置为我们想要在循环中调用的函数。我们首先要调用的就是`systemTitle`函数。

## systemTitle函数定义

`systemTitle`函数显示一个标题画面然后跳转到一个等待按钮的点击去关闭`BasicScreen`窗口的状态(`STATE_SYSTEM_WAIT_FOR_CLOSE`)。我们已经详细的阐述 `STATE_SYSTEM_TITLE`,

`STATE_SYSTEM_INSTRUCTIONS`, `SYSTEM_LEVEL_IN`, and `STATE_SYSTEM_GAME_OVER`这几种状态了，所以现在直接做就可以了。有两种基本的画面类型：

等待点击OK按钮的画面：`itleScreen`, `instructionsScreen`, 和`gameOverScreen`。

在移除前等待一个时间周期的画面：`levenInScreen`



首先，我们给OK按钮添加了一个事件监听器去监听点击事件，这个监听器命名为okButtonClickListener函数，并且将其作为所有的画面的共享监听。然后，我们选择systemState为STATE\_SYSTEM\_WAIT\_FOR\_CLOSE。然后我们设置的 nextSystemState就会被okButtonClickListener触发时切换。

```
titleScreen.addEventListener(CustomEventButtonId.BUTTON_ID,□  
okButtonClickListener,false, 0, true);
```

所有的BasicScreen实例都共享同样的okButtonClickListener函数。自定义事件传递的是BasicScreen实例的id值。Id值用来告诉事件是哪一个按钮被按下了并据此切换状态。

systemTitle, systemInstructions, 和 systemGameOver函数看起来非常相似。我们来检查一下他们的区别。首先，让我们检查waitForclose和okButtonClickListener函数。

## systemWaitForClose（系统等候关闭）函数定义

systemWaitForClose()函数与STATE\_SYSTEM\_WAIT\_FOR\_CLOSE状态相关联。它在点击OK按钮之前什么也不做。

这是你在本书中遇见的最简单的函数了。它绝对是什么也不做的！它只是作为游戏循环可以等待点击按钮的一个占位函数。

## okButtonClickListener（ok按钮点击监听）函数定义

okButtonClickListener函数用来在OK按钮在某个画面实例上被点击时进行判定。在触发后会切换到



nextSystemState ( 下一个状态 )。

这个“监听”函数只在收到一个CustomEventButtonId.BUTTON\_ID事件时被触发。

不管OK按钮是不是被点击，我们都要在切换状态时移除时间的监听。尽管同样的监听被用在标题画面和介绍画面中，我们都要在重新添加它之前先从标题画面中移除它，这样做是为了确保不会有任何的监听被遗忘移除。没用的监听浪费内存而且会导致流程变慢。

这个函数告诉我们如何给三个不同的画面（标题画面，介绍画面，和游戏结束的画面）给OK按钮的点击共享一个监听事件。我们根据CUSTOMEVENT\_OK\_CLICKED的id值进行切换。

不论在哪一个画面，最后一行的函数都会调用switchSystemState函数并传递nextSystemState变量。

## systemInstructions ( 系统介绍 ) 函数定义

systemInstructions函数用来给用户显示介绍画面。它与STATE\_SYSTEM\_INSTRUCTIONS状态相关。它调用一个单独的时间然后进入STATE\_SYSYEM\_WAIT\_FOR\_CLOSE状态。

systemInstructions函数跟systemTitle函数非常相似。实际上，他们只是稍微改动一点。我们首先用addChild把systemInstructionsScreen加入到显示列表。

然后加入一个跟systemTitle函数中使用的一样的CustomEvent（自定义事件），然后切换到STATE\_SYSTEM\_WAIT\_FOR\_CLOSE状态。同样的，这个状态除了接收BasicScreen类的实例中的CustomEventButtonId.BUTTON\_ID以外什么也不做。最后，我们在点击OK按钮后切换到nextSystemState中。

## systemGameOver ( 系统游戏结束 ) 函数定义

systemGameOver 函数显示 gameOverScreen（游戏结束画面）。它与



STATE\_SYSTEM\_GAME\_OVER状态相关联并与其他的基本Screen类实例一样需要等待点击OK按钮。

我们现在来看一看systemGameOver函数，因为它与titleScreen和instructionsScreen一样是BasicScreen的实例。当一个名为GAME\_OVER的自定义事件触发时Game类就进入systemGameOver状态。在下一个部分，我们会看见这个自定义事件的监听。

systemGameOver函数与systemTitle和systemInstructions函数的样式似乎是一样的。只有一处不一样的地方：它会通过removeChild函数将Game类的实例从显示列表上移除。

## systemNewGame（系统新游戏）函数定义

systemNewGame函数是与STATE\_SYSTEM\_NEW\_GAME状态相关联的。它调用一个计时器然后切换到STATE\_SYSTEM\_NEW\_LEVEL状态。它的目的就是加入所有在Game类和其他框架类（比如ScoreBoard类）通信的监听器。它也调用game.newGame函数来让Game的实例开始运行内部的新游戏进程。

当开始一个新游戏的时候，我们首先用addChild方法把我们的Game类实例加入显示列表。这会让Game.as作为Sprite（或者MovieClip）显示在屏幕上。然后，我们建立一些基本的Game.as和Main.as的联系。我们加入四个事件监听器来实现这一目的。前面的两个监听器是自定义事件类的实例（关于此类我们在后面的章节会详细谈）。

CustomEventLevelScreenUpdate类可以通过事件发送一个文本字符串。这个字符串可以是Game类传递给Main.as类的等级数字（或者任何文本）。然后Main.as类就可以根据传递的字符串来更新levelInString变量。我们很快就能看见这个监听函数了。

CustomEventScoreBoardUpdate类用来更新ScoreBoard（另一个类我们会在后面的章节中讨论）。这个事件把数据发送给scoreBoardUpdateListener并要求ScoreBoard将新数据更新。例如：日过我们想



更新玩家的分数，我们就要传回分数区域的名称（可能是“score”）和玩家分数的值（例如是 5000）。

我们还要创建两个名为GAME\_OVER和NEW\_LEVEL的事件常量。这些常量不传回任何数据只是作为触发语句给dispatchEvent函数。这种类型的事件我们就不需要用到自定义类了。

## systemNewLevel（系统新关卡）函数定义

systemNewLevel与STATE\_SYSTEM\_NEW\_LEVEL状态相关联。它的作用是调用game.newLevel函数从而让游戏开始新等级的游戏流程。

systemNewLevel函数在Main类中不做更多的事。仅仅是调用Game类的新Level函数罢了。这个newLevel函数我们会在本章的游戏实例中展示一下。它被用来初始化新关卡的变量和难度等级。函数运行完毕后，就会将系统状态切换至STATE\_SYSTEM\_LEVEL\_IN。

## systemLevelIn（系统关卡运行）函数定义

systemLevelIn()函数与STATE\_SYSTEM\_LEVEL\_IN状态相关联。它在 30 帧内显示一个新关卡的消息并将状态切换至STATE\_SYSTEM\_GAME\_PLAY。

使用systemLevelIn 并不是唯一能让状态机进入待机状态的方法。有一些第三方自定义工具类比如说TweenMax就可以用来进行往返缓动的同步。我们给状态机加入个等待状态是为了框架的完整性。状态机的设计理念就是可以用状态与系统函数进行简单的更新。体验一些免费的第三方工具类或者库让你的工作更简单。

systemLevelIn 函数可以让开发者在没关开始之前显示一些文字或者动画。这里需要的状态就是STATE\_SYSTEM\_WAIT。然后就会把levelInScreen加入到显示列表。levelInScreen是一个没有OK按钮的BasicScreen实例，而只是等待一个waitTime（等待时间，这里用 30 帧的时间来做例子）然后触发



WAIT\_COMPLETE这个自定义事件。Main类会监听这个事件然后调用相关的监听函数。这个画面所需的文本和 levelInText 变量都是我们在变量定义的部分中建立好的，文本会通过 CustomeEventLevelScreenUpdate进行传递。levelIntext可以在Main.as类中重载的init函数中被定义。它会接收事件发送过来的文本然后转给levelInScreen让其显示在画面上。我们马上就来看看这个监听的函数。

## systemWait ( 系统等待 ) 函数定义

levelInscreen与instructionsScreen非常相似，不同的是不用TATE\_SYSTEM\_WAIT\_FOR\_CLOSE状态，而是用STATE\_SYSTEM\_WAIT状态。有什么区别吗？STATE\_SYSTEM\_WAIT状态调用的是统计一个计算帧次的数字的函数而不是等待点击OK按钮的。当状态切换至STATE\_SYSTEM\_WAIT的时候，它反复的调用systemWait函数（每帧都调用一次）直到waitCount大于事先定好的waitTime为止。让waitCount的数值达到要求后，我们派遣一个WAIT\_COMPLETE事件来调用waitCompleteListener（下一个部分会讲解）。目前只有一个systemState需要用到systemWait函数，所以在这个状态的case语句中也只能看见这么一个东西。我们可以增加许多状态来使用systemWait函数，不过这要在后面的学习中再说了。

## waitCompleteListener ( 等待完成监听 ) 函数定义

waitCompleteListener函数在WAIT\_COMPLETE事件触发的时候被levelInScreen调用。它可以被用在很多状态下进行画面的刷新。一旦waitCompleteListener被触发，它就将状态切换至lastSystemState（上一个系统状态）因为currentSystemState（当前系统状态）是STATE\_SYSTEM\_WAIT。这使得我们可以给所有的WAIT\_COMPLETE事件共享一个监听函数。当它调用switchSystemState（切换系统状态）时候就把状态切换到nextSystemState（下一个系统状态）。在这个例子中，nextSystemState是



systemGameplay ( 系统游戏进行 )。

## systemGameplay() ( 系统游戏进行 ) 函数定义

systemGameplay 函数是游戏循环的心脏。它与 STATE\_SYSTEM\_GAME\_PLAY 状态相关联。

game.runGame 函数在每帧都被 systemGamePlay 状态下的 systemFunction 调用。Game 类的实例

( game ) 会在 game.runGame 函数中执行游戏的所有流程。

```
private function systemGameplay():void {  
  
    game.runGame();  
  
}
```

## 自定义事件监听函数

最后 GameFrameWork.as 类中的四个函数都是事件监听函数 , 它们的目的是在 Game 的实例与 Main 类的实例进行通信 , 嗯 , 还有 levelInScreen 和 scoreBoard 的实例。我们会在一个地方简短的将 Main.as 的代码给你完整的展示一下。这些监听都会后面的这些类中被使用 : Game , ScoreBoard 还有 CustomEvent 类。

## 为 Main.as 进行 scoreBoardUpdateListener ( 计分板更新监听 ) 函数定义

scoreBoardUpdateListener 接收从 Game 类实例的更新消息然后通过 CustomEventBoardUpdate 实例给 ScoreBorad 发送更新。

scoreBoardUpdateListener 传递的数据是两个字符串对象。其中包含一个元素变量 , 它包含一个预先在 Main.as 中定义的常量。常量是 scoreBoard 要更新元素的 id 名称。还有一个数值变量通过一个 TextFeild 将数值以字符串的形式展示在 scoreBorad 上。这样的结构可以让 Game 类实例和它的关联类之



间配合来更新ScoreBorad类实例而不用一直去引用它。这也使得Game和它的关联类得以和Main.as与基本框架类可以保持非耦合状态。我们会在稍后的levelInScreen中看见一样的情况。为什么要让ScoreBorad是框架的一部分却不是Game类的一部分？我们让ScoreBorad成为游戏框架中Main类的一部分的原因是Game类不是唯一要与ScoreBorad进行相互作用的唯一对象。我们想一想如果Main有其他的数据要传递，比如说系统最高分，一个链接，或者一个帧计时之类的东西要展示在ScoreBorad上的时候。

为什么要让计分板与游戏类实例非耦合？因为这样我们就可以直接从Game中用根级或者父级Game类中的方法调用ScoreBorad。或者想想别的情况，我们把ScoreBorad分离出来之后，你就可以把它放在任何一个游戏中去了，而不必拘泥于某个框架。

## 为Main.as进行levelScreenUpdateListener（关卡画面更新监听）函数定义

levelScreenUpdateListener像scoreBoardUpdateListener那样可以让Game类更新levelInScreen上的文本，levelScreenUpdateListener使用从Game类实例上传递过来的数据给levelInScreen。当Game类用其定义的新Level函数更新关卡值的时候，levelInScreen就跟着用systemLevelIn函数更新显示关卡数值。预定义的字符串变量与LevelInText是结合成一个参数值发送的：

```
levelInScreen.setText(levelInText + e.text);
```

## 为Main.as进行gameOverListener（游戏结束监听）函数定义

gameOverListener监听GameGAME\_OVER的自定义事件然后将框架的状态切换。

当Game类实例触发了GAME\_OVER自定义事件时，监听它的Main就运行一个函数。这个函数清除所有与游戏相关的监听器然后将状态切换至STATE\_SYSTEM\_GAME\_OVER。



## 为Main.as进行newLevelListener（新关卡监听）函数定义

newLevelListener函数监听Game.NEW\_LEVEL自定义事件并将状态切换。

当游戏类的实例通过NEW\_LEVEL参数触发newLevelListener自定义事件时，系统的状态就会切换至STATE\_SYSTEM\_NEW\_LEVEL。

## 深入Main框架类

框架并不是只靠FrameWorkStates和GameFrameWork类。我们现在要定义并说明BasicScreen类（还有他的辅助类），ScoreBoard类，还有自定义事件类。我们已经简略的讨论了GameFrameWork.as类的每一个部分。现在，我们来详细的看一看吧。

### BasicScreen类

所有的基本游戏框架中的简单画面都使用一个简单的BasicScreen类来建立。BasicScreen类可以作为将来更复杂画面的父类，但是在基本框架中，它碍事非常简单的。每个画面包含一些文本，一个按需添加的OK按钮和一种背景色。就这些了。如果需要一个OK按钮，相应的在BasicScreen类里就会有一个事件被触发，并告诉GameFrameWork类按钮是不是被点击了。这里要用到自定义事件类实例（okButtonClickListener）。我们会在下一个部分中创建它。

你应该把这个类文件保存在我们之前章节建立的框架包结构中。

/source/classes/com/efg/framework/BasicScreen.as

这里是类的代码：



```
package com.efg.framework

{

    // Import necessary classes from the flash library

    import flash.display.Bitmap;

    import flash.display.BitmapData;

    import flash.display.Sprite;

    import flash.events.Event;

    import flash.geom.Point;

    import flash.text.TextField;

    import flash.text.TextFormat;

    import flash.events.Event;

    import flash.events.MouseEvent;

    import flash.text.TextFormatAlign;

    /**
     * ...
     * @author Jeff Fulton, Steve Fulton
     */

    public class BasicScreen extends Sprite {

        private var displayText:TextField = new TextField();

        private var backgroundBitmapData:BitmapData;

        private var backgroundBitmap:Bitmap;
```



```
private var okButton:SimpleBlitButton;

//ID is passed into the constructor. When the OK button is
//clicked,a custom event sends this id back to Main

private var id:int;

public function BasicScreen(id:int,width:Number, height:Number,isTransparent:Boolean,
color:uint) {

    this.id = id;

    backgroundBitmapData = new BitmapData(width, height,IsTransparent, color);

    backgroundBitmap = new Bitmap(backgroundBitmapData);

    addChild(backgroundBitmap);

}

public function createDisplayText(text:String, □
width:Number, location:Point,textFormat:TextFormat):void {

    displayText.y = location.y;

    displayText.x = location.x;

    displayText.width = width;

    displayText.defaultTextFormat=textFormat;

    displayText.text = text;

    addChild(displayText);

}

public function createOkButton(text:String,location:Point, width:Number,height:Number,
```



```

textFormat:TextFormat,      offColor:uint=0x000000,      overColor:uint=0xff0000,

positionOffset:Number=0):void{

    okButton = new SimpleBlitButton(location.x, location.y, □

        width, height, text, 0xffffffff, 0xff0000, textFormat,□

        positionOffset);

    addChild(okButton);

    okButton.addEventListener(MouseEvent.MOUSE_OVER,□

        okButtonOverListener, false,0, true);

    okButton.addEventListener(MouseEvent.MOUSE_OUT,□

        okButtonOffListener, false, 0, true);

    okButton.addEventListener(MouseEvent.CLICK,□

        okButtonClickListener, false, 0, true);

}

public function setDisplayText(text:String):void {

    displayText.text = text;

}

//Listener functions

//okButtonClicked fires off a custom event and sends the

//"id" to the listener.

private function okButtonClickListener(e:MouseEvent):void {

    dispatchEvent(new CustomEventButtonId(CustomEventButtonId.BUTTON_ID,id));

```



```
}  
  
private function okButtonOverListener(e:MouseEvent):void {  
  
    okButton.changeBackgroundColor(SimpleBlitButton.OVER);  
  
}  
  
private function okButtonOffListener(e:MouseEvent):void {  
  
    okButton.changeBackgroundColor(SimpleBlitButton.OFF);  
  
}  
  
}  
  
}
```

## BasicScreen的类导入和变量定义

类导入部分包括必须的Flash核心类。BasicScreen类中的一个自定义类是CustomEventButtonId类。这个我们会在后面的章节详细说明，现在我们要派遣一个OK按钮在任何一个BasicScreen上是否被点击的事件。变量定义主要是为了保存三部分可选的数据：

- 背景色的BackGroundBitmapData，以及与其相关的BackGroundBitmap。
- 画面要显示的文本，displayText。
- 触发CustomEventButtonId点击事件的按钮，它是另一个名为SimpleBlitButton的框架自定义类实例。

这些东西都在BasicScreen类的public函数中进行设置。每个游戏的Main.as类都要通过给



BasicScreen的三个public函数传递相关的参数来定义初始化这些变量。

关于BasicScreen类有一个事是关于OK按钮的背景色的。我们决定在第一个游戏中不导入或者嵌入任何的外观资源，因为这样的话我必须给你解释它们在Flash IDE和Flash Develop（还有开源的Flex SDK）中不同的做法，而这对于目前的你来说还为时过早。因此，我们现在只是建立一个名为SimpleBlitButton的简单按钮。它的实例名称就叫okButton吧。我们会在SimpleBlitButton类定义的部分中进行说明。

最后我们需要的变量就是id整数了。这是从Main类中传递给BasicScreen实例的。它用来给GameFrameWork类里的okButtonClickListener函数中的switch:case部分进行识别是哪一个画面里的OK按钮被点击了。

## 构造函数定义

BasicScreen类的构造函数需要设置一些画面的基本信息。一个画面需要一个整形的id值，当然还有画面的背景颜色信息。这会在定义backGroundBitmapData变量的部分中介绍。一个BitmapData实例需要下面四个参数：

- 一个宽度值
- 一个高度值
- 一个布尔值（真或者假）来确定BitmapData是否要支持透明。
- 一个uint型的颜色值。

一个完整的透明背景需要将是否透明参数设置为true，之后需要一个32位带透明通道的颜色值。这个颜色值的格式是AARRBGGG，其中AA表示从0x00(0)(表示完全透明)到0xFF(255)(完全不透明)的透明信息。

Init函数用来显示文本域(displayText)和OK按钮(okButtonSprite)。如果需要一个按钮只需要将okNeeded参数设置为true就可以了。



不过第一行的代码看起来似乎有一点迷糊:

```
textformat1.align = flash.text.textFormatAlign.CENTER
```

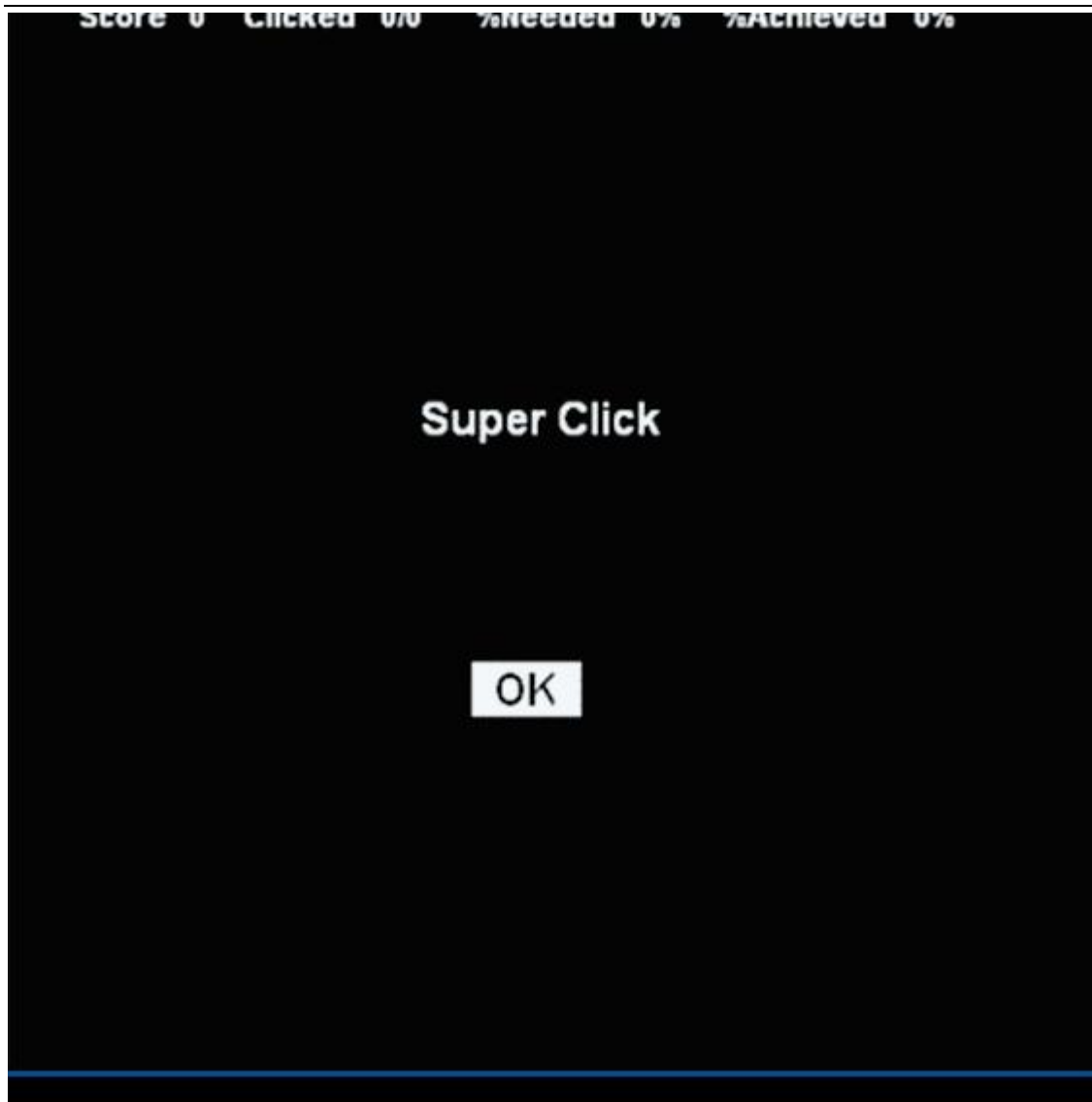
当我们设置textFormat1 变量的时候,我们不要先设置它的对齐方式因为不是所有的文字都需要居中对齐。在我们把文本应用其自身的defaultTextFormat(默认文本格式)之前,我们可以随时更改对齐方式。然后我们只要将其加到画面上就可以了。在init函数第一次调用的时候,文字实际上并没有被加入到画面里。它要在下次setDisplayText函数调用的时候才会被加入。setDisplayText函数是一个public函数因此可以在BasicScreen实例(标题画面,介绍画面,游戏结束画面,任何一个画面)加入到显示列表的时候被Main类调用。这让我们可以在每次显示画面的文字的时候进行自定义设置。

## createDisplayText ( 建立显示文本 ) 函数定义

通过在一个BasicScreen实例上调用createDisplayText函数,我们可以得知文本的位置与内容。然后他能够多addChild方法将其加入到显示列表中去。游戏画面如图示 2-1 中所示。

这个函数的参数如下:

- text:这个参数会改变TextField的.text属性。
- location:用一个Point类实例来定义文本域左上角注册点的x坐标和y坐标。
- width:宽度的单位是像素。
- textFormat:一个TextFormat实例用来设置TextFeild的文本格式。



图示 2-1 Super Click 带OK按钮的标题画面

## createOkButton ( 创建OK按钮 ) 函数定义

如果BasicScreen实例需要一个OK按钮来进行状态机的交互，那么就需要加入这个public的createOkButton函数了。这个函数做的事情并不多。首先，它要创建一个SimpleBlitButton类实例（这个类我们稍后再说）。SimpleBlitButton类需要一些参数给构造函数，然后用它们串讲一个带有OVER状态和OFF状态的按钮。OVER状态就是鼠标指针经过按钮的状态，OFF状态就是指针不在按钮上的状态。对于我们这个基本画面来说，我们只需要一个简单的，可重用的OK 按钮。它需要在OFF状态有一个黑色的文字和白色的背景色，在OVER状态则是一个黑色的文字和红色的背景色。最后这个函数就给这个按钮



实例添加一个鼠标事件监听。

这个函数的参数如下：

- text:按钮上方的文字
- location:用一个Point类实例来设置按钮注册点的坐标。
- width:按钮的宽度。
- height:按钮的高度。
- textFormat:按钮文字的字体格式。
- offColor:按钮在OFF状态下的背景色。
- overColor:按钮在over状态下的背景色。
- positionOffset:让按钮的文字在两种状态切换的时浮动效果的距离。

## setDisplayText(设置显示文本)函数定义

这个函数用来重置显示文本的.text属性，通过传递一个字符串即可。这通常用来设置levelInScreen上游戏关卡数字等文本。

## 按钮监听函数定义

okButtonClickListener: 这个事件派遣一个CustomEventButtonId实例和id值给监听器。在GameFrameWork类中，我们已经设置了这个事件的监听器。

okButtonOverListener: 这个事件调用SimpleBlitButton的changeBackgroundColor函数并发送OVER常量。然后SimpleBlitButton就会将按钮的背景色切换至OVER状态下的背景色。

okButtonOverListener:这个事件调用SimpleBlitButton的changeBackgroundColor函数并发送OFF常量。然后SimpleBlitButton就会将按钮的背景色切换至OFF状态下的背景色。

## SimpleBlitButton ( 简单图形按钮 ) 类

SimpleBlitButton类可以用来创建一个UI非常简单的按钮。它只有一个背景色 ( 实际上是两个 , OFF状态和OVER状态下各一种 ), 和一个文本标签。它用BitmapData存储颜色信息和Bitmap绘制的方法类



建立背景色。之所以叫图形按钮就是因为我们使用的是这种简单的Bitmap图形绘制技术。这样我们可以在需要的时候改变按钮的背景色。而且，我们还可以将文字也绘制进BitmapData里来动态的创建按钮。

术语“图形化”或者“图块”来自古典游戏开发方法。图形在内存中运行的速度非常快，因此，图形位移技术可以用来作为精灵图片显示，玩家导弹图形绘制，很大的二进制数组或者任何这种大量数据的组合。我们用“图形化”来表述这种用 *Bitmap* 和 *BitmapData* 方法绘制各种图形的意思。

在我们建立框架包结构中创建SimpleBlitButton.as类文件：

/source/classes/com/efg/framework/SimpleBlitButton.as

下面是这个类的全部代码：

```
package com.efg.framework
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.text.TextFormat;
    import flash.text.TextField;

    /**
     * ...
     * @author Jeff Fulton and Steve Fulton
     */
}
```



```
public class SimpleBlitButton extends Sprite {  
  
    public static const OFF:int = 1;  
  
    public static const OVER:int = 2;  
  
    private var offBackGroundBD:BitmapData;  
  
    private var overBackGroundBD:BitmapData;  
  
    private var positionOffset:Number;  
  
    private var buttonBackGroundBitmap:Bitmap;  
  
    private var buttonTextBitmapData:BitmapData;  
  
    private var buttonTextBitmap:Bitmap;  
  
    public function SimpleBlitButton(x:Number,y:Number,width:Number,height:Number,text:String, offColor:uint, ☐  
overColor:uint, textformat:TextFormat,positionOffset:Number=0) {  
  
        this.positionOffset = positionOffset;  
  
        this.x = x;  
  
        this.y = y;  
  
        //background  
  
        offBackGroundBD = new BitmapData(width, height, false, offColor);  
  
        overBackGroundBD = new BitmapData(width, height,false, overColor);  
  
        buttonBackGroundBitmap = new Bitmap(offBackGroundBD);  
  
        //text  
  
        var tempText:TextField = new TextField();
```



```
tempText.text = text;

tempText.setTextFormat(textformat);

buttonTextBitmapData = new BitmapData(
    (tempText.textWidth+positionOffset, tempText.
    textHeight+positionOffset, true, 0x00000000);

buttonTextBitmapData.draw(tempText);

buttonTextBitmap = new Bitmap(buttonTextBitmapData);

buttonTextBitmap.x = ((buttonBackGroundBitmap.width -
    int(tempText.textWidth))/2)-positionOffset;

buttonTextBitmap.y = ((buttonBackGroundBitmap.height -
    int(tempText.textHeight))/2)-positionOffset;

addChild(buttonBackGroundBitmap);

addChild(buttonTextBitmap);

this.buttonMode = true;

this.useHandCursor = true;
}

public function changeBackgroundColor(typeval:int):void {

    if (typeval == SimpleBlitButton.OFF) {

        buttonBackGroundBitmap.bitmapData = offBackGroundBD;

    }else {

        buttonBackGroundBitmap.bitmapData = overBackGroundBD;
```



```
}  
  
}  
  
}  
  
}
```

## SimpleBlitButton类的类导入和变量定义

我们创建的按钮实际上是一个含有两个 Bitmap 图层的 Sprite 容器。底层是 buttonBackGroundBitmap。它的 BitmapData 属性可以在 offBackGroundBD 和 overBackGroundBD 之间进行切换。按钮的文本位于最上方。它包含 buttonTextBitmapData 的 bitmapData 实例。我们定义两个常量来表示按钮的两个状态。

```
public static const OFF:int = 1;
```

```
public static const OVER:int = 2;
```

我们在这个类的狗仔函数创建按钮的文本。buttonTextBitmapData 将文字绘制进要显示的 BitmapData 中去。然而我们可以直接用 TextField 放进 Sprite 按钮里，这是本书游戏中的一个重点。因为我们想尽早的让你接触利用 Bitmaps 和 BitmapData 来做动画。我们还会集中在图形技术上，因为我们想让你在这部分中就了解用图形技术做动画。你越早了解 BitmapData 的多种用法，你在其余章节中的讲解中越轻松。

我们还要定义另一个变量。它在需要的时候可以让文字 “bump ( 凸起？ )”。在一些字体下你可能发现文字从右边被缩短了。buttonTextBitmapData 变量就可以帮助你缓解这个问题。



## SimpleBlitButton类的构造函数定义

SimpleBlitButton类的构造函数参数必须包括正确的尺寸,颜色和按钮的文本以及其两个状态:OFF和OVER状态。

构造函数需要这些参数:

- 画面位置的x值
- 画面位置的y值
- 按钮的宽度
- 按钮的高度
- 按钮上文本标签的字符串
- 按钮在OFF状态下的背景色 ( uint型 )
- 按钮在OVER状态下的背景色 ( uint型 )
- 一个设置按钮文本格式的TextFormat对象
- 用来修正文本在bump时位置偏移的positionOffset值

设置好按钮的x、y坐标值后,我们就来用BitmapData对象创建按钮的两种背景色。而BitmapData需要获取宽度、高度以及颜色值来创建位图。buttonBackGroundBitmap就是用来存储绘制的位图用的,默认的情况下,它显示的是OFF状态下的图像。

```
buttonBackGroundBitmap = new Bitmap(offBackGroundBD);
```

背景色创建好之后,我们来看看如何生成按钮的文字。我们的目的就是把文字显示在背景图的中央。

我们用一个临时文本 ( tempText ) TextFeild来设置文字的格式。



在这个简单的按钮里，我们不用检查文字是否适合按钮，所以如果文字的宽高不合适的话，它就会被裁剪。BitmapData可以让文字不会溢出它的矩形区域，因为矩形以外的地方不是可是区域（这一点不想Stage或者Sprites和MovieClips）。最后我们用临时格式（tempFormat）来设置文字的字体格式。

现在我们要把文字加到按钮的中央。这需要一点测量，因为我们会发现有时候文本域的边缘有两个像素左右的差值。我们用positionOffset来纠正这一点。我们会传递文字的宽高（textWidth值和textHeight值），一个透明的背景（true）和一个不透明的背景色（0x00000000）。

```
buttonTextBD = new BitmapData(tempText.textWidth,□  
tempText.textHeight, true, 0x00000000);
```

0x00000000 是一个 32 位AARRBGGG格式的颜色值。AA就是透明度。我们设置成 00，就是建立一个看不见的背景色，不论是什么颜色（这里用的是黑色）。

获取tempText.text值给buttonTextBitmapData和BitmapData实例，我们调用draw方法把文字绘制进buttonTextBitmapData中：

```
buttonTextBitmapData.draw(tempText);
```

draw方法可以把任何显示对象用像素绘制在BitmapData对象里。Draw方法可以用来获取任何矢量对象（或者位图对象）的内容并放在缓存中。这个方法是出了名的低效，但是我们不会再游戏中频繁的使用。我们用它建立稍后要用到的对象。

最后我们必须让文字的按钮加入到一个显示对象中。我们已经创建一个用来保存文字的位图的实例



叫buttonTextBitmap。这个显示对象会放在buttonBackGroundBitmap的顶层来作出在OFF和OVER状态的点击效果。这行代码创建了 buttonTextBitmap :注意我们传进去的实例是buttonTextBitmapData的实例。这会成为buttonTextBitmapData的bitmapData属性。

```
buttonTextBitmap = new Bitmap(buttonTextBD);
```

当我们把文字放在按钮的中间时，我们会把文字的左上方偏移 2 像素再绘制进 buttonTextBitmapData。我们可以用matrix方法来改变文字的位置，但为了简化，我们会让x坐标和y坐标移动 2 像素来达到这一效果。我们像这样计算中心的坐标值：

```
//X: (the background width - the text width)/2 minus the 2pixel buffer
```

```
//Y: (the background height - the text height)/2 minus
```

```
//the 2pixel buffer
```

```
buttonTextBitmap.x = ((buttonBackGroundBitmap.width - int(tempText.textWidth))/2)-2;
```

```
buttonTextBitmap.y = ((buttonBackGroundBitmap.height - int(tempText.textHeight))/2)-2;
```

接下来我们要让按钮的buttonBackGroundBitmap和buttonTextBitmap按顺序加入到这个Sprites容器中。最后就是设置按钮的手形模式。

## changeBackgroundColor ( 改变背景颜色 ) 函数定义

changeBackgroundColor用来改变按钮的在OFF状态和OVER状态时的背景色。它需要的常量参数分别代表了我们在构造函数中创建的两种按钮状态的BitmapData实例。



## CustomEventButtonId ( 自定义事件按钮ID ) 类

CustomEventButtonId类用来在事件触发时给监听函数传递数据。它继承了Flash内置Event类，所以相对来说较为简单。它的主要函数就是根据传递的整型值派遣事件。

把 CustomEventButtonId 类文件建立在我们之前创建的包结构中：

/source/classes/com/efg/framework/CustoEventButtonId.as

下面是这个类的全部代码：

```
package com.efg.framework
{
    import flash.events.*;

    /**
     * ...
     * @author Jeff Fulton
     */
    public class CustomEventButtonId extends Event {

        public static const BUTTON_ID:String = "button id";

        public var id:int;

        public function CustomEventButtonId(type:String,id:int, bubbles:Boolean=false,
cancelable:Boolean=false){

            super(type, bubbles,cancelable);
```



```
this.id = id;

}

public override function clone():Event {

return new CustomEventButtonId(type,id, bubbles,cancelable)

}

}

}
```

## CustomEventButtonId类的类导入与变量定义

在这部分里，我们设置类需要的变量。我们会创建一个按钮ID事件需要的常量。

```
BUTTON_ID:String = "button id"
```

Id字符串变量用来在事件与监听器之间进行传递。

我们传递BasicScreen画面上被点击的OK按钮的id值。如果按钮在一个BasicScreen上，id（当我们初始化BasicScreen实例的时候会被定义）就会传递给多有的事件监听。在框架中，主要的监听都是用来检测状态切换与按钮点击的。

## CustomEventButtonId类的构造函数定义

构造函数中第一个字符串应该就是我们之前定义的BUTTON\_ID常量了。你可以创建一个



CustomEventButtonId实例并用这种方式传递一个字符串给它的构造函数。传递一个常量（你可以增加更多的）比传递一个字符串更有组织性。这还可以避免一些未知的运行错误。如果你的监听函数触发了一个名为GAME\_OVER的事件但是又打错了GAME\_OVER，你可能就要艰难的调试一个令人讨厌的事件属性错误。

让我们再看看BasicScreen是怎么声明一个CustomEvent类的吧。

```
dispatchEvent(new CustomEvent(CustomEventButtonId.BUTTON_ID,id));
```

当事件被创建时，我们把CustomEventButtonId.BUTTON\_ID常量传递进去以触发相应的事件，然后还要传递一个派遣这个事件的BasicScreen实例的id。

GameFrameWork.as是怎么监听一个自定义事件的？主要需要给所有带有OK按钮的画面上加入监听器。这些监听器在需要的时候就加上不需要的时候就删除掉。让我们来回顾一下GameFrameWork是怎么给一个标题画面加入这个事件的吧：

```
titleScreen.addEventListener(CustomEventButtonId.BUTTON_ID,okButtonClickListener, false, 0, true);
```

这样，我们就可以给任何实例加入一个可以调用CustomEventButtonId事件的CustomEventButtonId实例。通过这种方式，所有的画面都可以用同样的监听器触发同样的事件。我们已经快速浏览了okButtonClickedListener函数。它用事件的id值判断是哪个按钮被点击并进行状态的切换。



```
super(type, bubbles,cancelable);
```

this.id变量被传进来的id重新赋值。

```
this.id = id;
```

CustomEventButtonId类的clone（克隆）函数定义

Clone函数是内置事件类的一部分。因此它必须用override加以声明，因为自定义事件需要的属性必须加入到clone函数中。

```
public override function clone():Event {  
  
    return new CustomEvent(type,id, bubbles,cancelable)  
  
}
```

Clone函数本来是事件类的标准函数，但是我们要覆盖它来保证传进去的属性可以返回一个新的自定义事件。属性对象都是我们自己创建的。

## CustomEventLevelScreenUpdate类

CustomEventLevelScreenUpdate事件类跟CustomEventButtonId类非常相似。它用来传递一个新文本值给监听函数。特别的地方就是它是GameFrameWork.as用来在新关卡的时候更新文本给leveininScreen。



把CustomEventLevelScreenUpdate.as类文件建立在我们之前创建的包结构中：

/source/classes/com/efg/framework/CustoEventLevelScreenUpdate.as

下面是这个类的全部代码：

```
package com.efg.framework
{
    import flash.events.Event;

    /**
     * ...
     * @author Jeff Fulton
     */
    public class CustomEventLevelScreenUpdate extends Event {

        public static const UPDATE_TEXT:String = "update level text";

        public var text:String;

        public function CustomEventLevelScreenUpdate(type:String,
            text:String,bubbles:Boolean=false,cancelable:Boolean=false) {

            super(type, bubbles,cancelable);

            this.text = text;
        }

        public override function clone():Event {

            return new CustomEventLevelScreenUpdate(type,text,
                bubbles,cancelable)
```



```
}  
  
}  
  
}
```

三种自定义事件中真正不同地方就是我们创建实例的时候常量值还有给触发函数传递的数据了。

```
public static const UPDATE_TEXT:String = "update level text";  
  
public var text:String;
```

所以我們也需要修改构造函数和clone函数类让新类型的数据可以传递。

## CustomEventScoreBoardUpdate类

这个自定义事件类与CustomEventButtonId类和CustomEventLevelScreenUpdate类非常相似。

它传递两个字符串值给监听函数。它用来给ScoreBoard类实例更新文本。

把CustomEventScoreBoardUpdate.as类文件建立在我们之前创建的包结构中：

```
/source/classes/com/efg/framework/CustoEventScoreBoardUpdate.as
```

下面是这个类的全部代码：

```
package com.efg.framework  
  
{  
  
    import flash.events.Event;  
  
    /**  
  
    * ...
```



```
* @author Jeff Fulton
*/

public class CustomEventScoreBoardUpdate extends Event {

    public static const UPDATE_TEXT:String = "
    \"update scoreboard text\";

    public var element:String;

    public var value:String;

    public function CustomEventScoreBoardUpdate(type:String,element:String, value:String,
    \"
    bubbles:Boolean=false,cancelable:Boolean=false) {

        super(type, bubbles,cancelable);

        this.element = element;

        this.value = value;

    }

    public override function clone():Event {

        return new CustomEventScoreBoardUpdate(type,element,value, bubbles,cancelable)

    }

}

}
```

不同的是就是事件的常量和传递的数据类型：



```
public static const UPDATE_TEXT:String =  
"update scoreboard text";  
public var element:String;  
public var value:String;
```

Element变量与Main.as ( GameFrameWork.as的子类 ) 中的计分板的element相对应。Value变量就是element要改变的值。现在我们跳进ScoreBoard框架类来让这部分完整起来。

## ScoreBoard 类

ScoreBoard类在你创建每个游戏时都需要一些改变。像BasicScreen类，ScoreBoard类在框架中都是非常简单的，而且你可能会在以后的游戏中进行一些扩展和增强。尽管如此还是有些基本的功能是所有的计分板需要的，比如事件和在它的game类实例之间的通信。ScoreBoard让Main.as可以创建一个有许多SideBySideScoreElement类（下一部分会讲到）实例的组合对象。这些对象在Main.as类中会被命名。让我们看看这个类的代码，然后你看到在我们创建StubGame的时候Main.as是如何创建这个计分板的。

把ScoreBoard.as类文件建立在我们之前创建的包结构中：

```
/source/classes/com/efg/framework/ScoreBoard.as
```

下面是这个类的全部代码：

```
package com.efg.framework  
{  
  
    // Import necessary classes from the flash libraries
```



```
import flash.text.TextField;

import flash.text.TextFormat;

import flash.display.Sprite;

/**
 * ...
 * @author Jeff Fulton and Steve Fulton
 */

public class ScoreBoard extends Sprite {

    private var textElements:Object;

    //Constructor calls init() only

    public function ScoreBoard() {

        init();

    }

    private function init():void {

        textElements = {};

    }

    public function createTextElement(key:String,obj:SideBySideScoreElement):void {

        textElements[key] = obj;

        addChild(obj);

    }

    public function update(key:String, value:String):void {
```



```
var tempElement:SideBySideScoreElement = textElements[key];  
  
tempElement.setContentText(value);  
  
}  
  
}  
  
}
```

## ScoreBoard类的类导入与变量定义

这个版本的ScoreBoard类放置文本标签并让它们按顺序水平排列在画面上。只要我们在任何BasicScreen实例上方通过SimpleBlitButton类建立一个图形按钮的时候，我们就会创建一个名为SideBySideScoreElement的类作为辅助类。

textElements对象是一个保存所有SideBySideScoreElement类实例的数组。

然后我们定义SideBySideScoreElement实例来保存计分板的UI对象。

ScoreBoard归属于GameFrameWork类，而不是Game类。实际上，我们要在这里说一下“设计模式”的概念，ScoreBoard类和Game类之间除非通过GameFrameWork类否则就不能建立通信。我们这么做可以保持类的整洁并让将来进行其他外来的引用变得困难（译者注：我觉得应该是容易才对）。对于框架来说，GameFrameWork类可以说是一个管理类。Game类会尽可能的与Main.as保持相对独立。我们要让游戏的逻辑完全与其他框架类脱藕。这么做的话就可以让框架的游戏逻辑可以拿出来放到其他框架中（如果需要的话）。

构造函数和init函数跟随的这种形式我们几乎在所有的自定义类中使用（除了几个辅助类）。如果构造函数没有参数传递进来，我们通常直接调用init函数来设置自定义类。

Init 函数 初始化 textElements 对象 并 保存 SideBySideScoreElement 类 实例 。



SideBySideScoreElement类我们会在下一个部分讨论。当我们在本章的定制GameFrameWork.as类（在游戏工程中这个文件叫Main.as）中，我们调用了一个名为SideBySideScoreElement的public函数。它创建了一个存储textElement对象并可以通过key访问的关联数组。Key是一个我们传给SideBySideScoreElement类构造函数的一个字符串。例如，字符串“score”会访问SideBySideScoreElement并返回score给使用者。

## 调用createTextElement需要的元素：

- 一个字符串作为key，这有点像BasicScreenClass中的id。
- 一个SideBySideScoreElement类的实例。

创建一个SideBySideScoreElement类实例需要八个参数。这个类创建一个静态标签的区域和一个动态文本区域。SideBySideScoreElement类会在下一部分详细说明。现在，你要了解如何在ScoreBoard上创建一个显示字符串标签和其动态字符的元素。它们之间要用一个空格分离。例如，我们需要给分数建立一个SideBySideScoreElement实例。标签的单词就回是Score然后跟着是玩家的分数。

- 标签和文本组合的x坐标。
- 标签和文本组合的y坐标。
- 标签和文本之间空隙的大小，单位是像素。
- 标签的文本字符串，比如“Score”。
- 标签的字体格式。
- 标签的宽度。
- 内容文本的初始字符串，比如在开始一个游戏的时候就是0。



- 内容文本的字体格式。

## createTextElement (创建文本元素) 函数定义

createTextElement存储传递进来的key值并据此将关联数组中对应的SideBySideScoreElement实例更新。这写会在被Main调用的更新函数中用来更新分数元素。

```
public function createTextElement(key:String,□  
obj:SideBySideScoreElement):void {  
  
    textElements[key] = obj;  
  
    addChild(obj);  
  
}
```

## 更新函数定义

更新函数是一个public函数。它可以更新ScoreBorad上的SideBySideScoreElement实例。在框架中，Main类在每次从Game类接收事件实例的时候调用。

```
//update() is called by Main after receiving a custom event  
  
//from the Game class  
  
tempElement.setContentText(value);
```

当分数需要更新的时候我们就用CustomEventScoreBoardUpdate触发事件。

```
dispatchEvent(new
```



```
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_
BOARD_CLICKS,String(clicks)));
```

GameFrameWork从CustomEventScoreBoardUpdate监听ScoreBorad类的更新。

```
game.addEventListener(CustomEventScoreBoardUpdate.UPDATE_TEXT,
scoreBoardUpdateListener, false, 0, true);
```

GameFrameWork 类要作用的这些事件在 scoreBoardUpdateListener 函数中需要给 ScoreBoard.update 方法传递值。函数获取传递进来的 key 值作为参数来电泳相应的 SideBySideSoreElement实例更新方法。详细的SideBySideSoreElement类在下部分讲解。

## SideBySideScoreElement类

SideBySideScoreElement就是我们成为辅助类的东西。它可以不用或者使用其他游戏元素，但是它的主要目标是简化ScoreBorad类的结构和代码。假如我们在写一个框架的时候，我们发现一些类的代码很多都是复制过来的。这通常是一个危险信号，那意味着这个类做了太多而且需要改进或者分割成一些辅助类。我们创建的其它辅助类就是BasicScreen类的CustomBlitButton类。

把SieBySideScoreElement.as文件建立在我们之前创建的包结构中：

```
/source/classes/com/efg/framework/SideBySideScoreElement.as
```

下面是这个类的全部代码：

```
package com.efg.framework
```



```
{  
  
import flash.display.Sprite;  
  
import flash.text.TextField;  
  
import flash.text.TextFormat;  
  
/**  
 * ...  
 * @author Jeff Fulton  
 */  
  
public class SideBySideScoreElement extends Sprite {  
  
    private var label:TextField = new TextField();  
  
    private var content:TextField = new TextField();  
  
    private var bufferWidth:Number;  
  
    public function SideBySideScoreElement(x:Number, y:Number,□  
        bufferWidth:Number, labelText:String, labelTextFormat:□  
        TextFormat, labelWidth:Number, contentText:String,□  
        contentTextFormat:TextFormat) {  
  
        this.x = x;  
  
        this.y = y;  
  
        this.bufferWidth= bufferWidth;  
  
        label.autoSize;  
  
        label.defaultTextFormat = labelTextFormat;  

```



```
        label.text = labelText;

        content.autoSize;

        content.defaultTextFormat = contentTextFormat;

        content.text = contentText;

        label.x = 0;

        content.x = labelWidth + bufferWidth;

        addChild(label);

        addChild(content);

    }

    public function setLabelText(str:String):void {

        label.text = str;

    }

    public function setContentText(str:String):void {

        content.text = str;

    }

}

}
```

什么样才是一个完善的计分板元素？比如说我们要显示玩家的分数吧。我们有一个Score标签后面跟着实际的分数值。在画面上我们要让它看起来是这样的：Score 850。

虽然这不是什么很酷的效果，但是它对于本书的简单游戏来说正合适。

这个类的变量定义部分非常简单。我们需要定义一个变量来保存标签和内容文本值以及这两个元素



之间的分隔符。

这个类继承了Sprite并包含两个文本域，分隔符的大小用bufferWidth变量来表示。标签变量保存的文本显示出来的是一个左对齐的静态标签。内容文本则是一个右对齐的标签。所以如果按照之前的例子来说，单词Score就是标签，850 这个值就是内容。

## 构造函数定义

构造函数做了本类中最多的活。这里就是需要传递的参数定义：

- x是标签和文本在ScoreBorad上开始绘制的x坐标。
- y是标签和文本在ScoreBorad上开始绘制的y坐标。
- bufferWidth是标签和文本之间的分隔效果，单位是像素。
- labelText是标签的字符串。
- labelTextFormat是标签文本的字体格式。
- labelWidth是标签的宽度。
- contentText是初始化的内容文本。
- contentTextFormat是内容文本的字体格式。

构造函数把标签的文本域实例和内容的文本域实例根据参数进行适当的安排。内容文本域的x坐标就是标签域的宽度值加上分割效果的数值。这让标签和内容文本在画面上错落有致。至于其他的属性都是简单明了的，不多说了。

## setLabelText和setContentText定义函数



这两个函数都是public函数，它们让标签文本和内容文本可以被ScoreBoard类改变。我们之前说过标签文本是静态的，但是既然没办法从一个动态文本域真正的创建静态文本，我们决定让标签可以通过setLabelText函数改变，以便之后的需要。

```
public function setLabelText(str:String):void {  
  
    label.text = str;  
  
}  
  
public function setContentText(str:String):void {  
  
    content.text = str;  
  
}
```

## Game类

Game.as框架类是所有利用框架建立的游戏的父类。它包括了与GameFramework的子类Main进行交互的基本功能。

把Game.as类文件建立在我们之前创建的包结构中：

/source/classes/com/efg/framework/Game.as

这里是本类的全部代码：

```
package com.efg.framework  
  
{  
  
    // Import necessary classes from the flash libraries  
  
    import flash.display.MovieClip;
```



```
import com.efg.framework.CustomEventScoreBoardUpdate;

import com.efg.framework.CustomEventLevelScreenUpdate;

/**
 * ...
 * @author Jeff Fulton
 */

public class Game extends MovieClip {

    //Create constants for simple custom events

    public static const GAME_OVER:String = "game over";

    public static const NEW_LEVEL:String = "new level";

    //Constructor calls init() only

    public function Game() {}

    public function newGame():void {}

    public function newLevel():void {}

    public function runGame():void {}

}

}
```

Game.as类是一个空壳，它仅仅包含了子类继承时必要的函数和与GameFrameWork.as类交互中需要的常量。在每个游戏的Main.as类中，我们会重新定义这些方法。

GAME\_OVER 和 NEW\_LEVEL 这两个常量，用来在 GameFrameWork 类中进行状态切换。GameFrameWork类的监听这两个事件的方法我们已经在之前的部分中见过了。



我们还声明了 newGame , newLevel 和 runGame 函数 , GameFrameWork 会对应的调用 systemNewGame , systemNewLevel 和 systemGamePlay 函数。让我们现在就来创建 stub 游戏来展示如何建立 GameFrameWork.as 的子类 Main.as 和 Game.as 的子类 StubGame.as。

## stub 游戏类

我们即将着手本书中第一个用框架类搭建起来的游戏。虽然这不是一个多么给力的游戏但是对你测试框架的代码是有帮助的。这可以让你确信整个框架是没问题的 , 然后才可以进行第三章的 Super Click 游戏。

## 游戏目标

点击鼠标十次。就这些。第一章我们做过同样的游戏。

## 游戏状态画面流程

我们的游戏需要四个 BasicScreen 实例。其中三个都带有 SimpleBlitButton 实例来切换游戏的状态。这些画面分别是 titleScreen , instructionsScreen 和 gameOverScreen 。最后的一个画面是 levelInScreen。这个画面没有 SimpleBlitButton 实例来切换游戏状态。这四个 BasicScreen 实例已经在 GameFrameWork 类中创建好了。我们要创建一个 Main.as 类给我们的游戏 , 它会继承 GameFrameWork.as 类并为了我们的 stub 游戏设置这些画面。

## 游戏 ScoreBoard ( 计分板 ) 常量

我们需要创建一个含有单独元素的计分板 , 这个唯一的元素来保存玩家点击鼠标的此时。在 Main.as 类中我们声明这个常量。这个元素的常量叫做 SCORE\_BOARD\_CLICKS。



## 创建Main.as类

让我们开始干真格的吧。在接下来的几页中，你会创建你的第一个用框架建立的游戏并且之后还会创建本书中其余的游戏。我们之前给两种不同的开发环境都做了工程结构。如果你还没有创建Main.as类文件，现在就去做。你需要把Main.as类文件保存在你的开发环境中的工程文件夹中：

在Flash IDE中，你的文件夹结构是这样的：

[source]

[projects]

[stubgame]

[flashIDE]

[com]

[efg]

[games]

[stubgame]

Main.as

在Flex SDK ( 或者是Flash Develop ) 中，结构则是这样：

[source]

[projects]

[stubgame]

[flexSDK]

[bin]



[obj]

[lib]

[src]

[com]

[efg]

[games]

[stubgame]

Main.as

下面是本类的全部代码：

```
package com.efg.games.stubgame  
  
{  
  
    import flash.text.TextFormat;  
  
    import flash.text.TextFormatAlign;  
  
    import flash.geom.Point;  
  
    import com.efg.framework.FrameWorkStates;  
  
    import com.efg.framework.GameFrameWork;  
  
    import com.efg.framework.BasicScreen;  
  
    import com.efg.framework.ScoreBoard;  
  
    import com.efg.framework.SideBySideScoreElement;  
  
    public class Main extends GameFrameWork {
```



```
//custom score board elements

public static const SCORE_BOARD_CLICKS:String = "clicked";

public function Main() {

    init();

}

override public function init():void {

    game = new StubGame();

    setApplicationBackGround(400, 400, false, 0x000000);

    //add score board to the screen as the seconf layer

    scoreBoard = new ScoreBoard();

    addChild(scoreBoard);

    scoreBoardTextFormat= new TextFormat("_sans", "11", "0xffffffff", "true");

    scoreBoard.createTextElement(SCORE_BOARD_CLICKS, □

    new SideBySideScoreElement(25, 5, 15, "Clicks", □

    scoreBoardTextFormat, 25, "0",scoreBoardTextFormat));

    //screen text initializations

    screenTextFormat = new TextFormat("_sans", "16", "0xffffffff", "false");

    screenTextFormat.align = flash.text.TextFormatAlign.CENTER;

    screenButtonFormat = new TextFormat("_sans", "12", "0x000000", "false");

    titleScreen = new

    BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE,400,400,false,0x0000dd );
```



```
titleScreen.createOkButton("OK", new Point(170, 250), 40, 20, screenButtonFormat,
0x000000, 0xff0000,2);

titleScreen.createDisplayText("Stub Game", 100,new
Point(145,150),screenTextFormat);

instructionsScreen = new
BasicScreen(FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS,400,400, false,0x0000dd);

instructionsScreen.createOkButton("Play", new Point(150, 250), 80,
20,screenButtonFormat, 0x000000, 0xff0000,2);

instructionsScreen.createDisplayText("Click the mouse\n10 times",150,new
Point(120,150),□
screenTextFormat);

gameOverScreen = new
BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER,400,400,false,0x0000dd);

gameOverScreen.createOkButton("OK", new Point(170, 250),40,
20,screenButtonFormat, 0x000000, 0xff0000,2);

gameOverScreen.createDisplayText("Game
Over",100,newPoint(140,150),screenTextFormat);

levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN, 400,
400, true, 0xaaff0000);

levelInText = "Level ";

levelInScreen.createDisplayText(levelInText,100,new
```



```
Point(150,150),screenTextFormat);

//Set standard wait time between levels

waitTime= 30;

//set initial game state

switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);

//create timer and run it one time

frameRate = 30;

startTimer();

}

}

}
```

## Main类的类导入和变量定义

除了需要的文本类和字体格式类，我们还导入了Point类，这样我们可以传递一个Point实例作为BasicScreen和SimpleBlitButton实例的坐标，然后我们可以在init函数中初始化它们。我们只需要给ScoreBoard定义一个常量。因为在我们的stub游戏里的计分板上只需要一个元素，所以我们只创建这么一个变量：

```
public static const SCORE_BOARD_CLICKS:String = "clicked";
```

注意Main.as继承了GameFrameWork类。这让我们可以直接访问或调用GameFrameWork里的所有public方法和属性。我们需要覆盖GameFrameWork.as的init函数来给每个不同的游戏初始化自定义



画面和计分板。

## 应用构造函数和init函数定义

Main.as类的构造函数仅调用init函数。在第 12 章，我们会创建一个在Flash IDE和Flex SDK中都可用的预加载器。预加载器的代码功能就是让进程等待舞台所有对象可用的时候再进入流程。不过在这个游戏里我们不需要，所以只要调用init函数就可以了。

Main.as类的核心就是覆盖GameFrameWork.as类文件的init函数。这里，我们会修改框架给每个单独的游戏。你完全的理解这个函数是非常重要的，因为它的修改和使用贯穿了本书后面的所有内容。

## 创建我们的游戏实例

Game变量在GameFrameWork.as文件中定义过了，但是要在Main.as中初始化。我们会在下个部分创建名为StubGame.as的Game类的子类。

```
game = new StubGame();
```

## 设置应用的背景色

appBackGroundBitmapData 在 GameFrameWork.as 类文件中定义，在 Main.as 中通过调用 appBackGroundBitmapData 的 public 函数进行设置。对于我们的 stub 游戏来说，我们要创建一个 400\*400 大小的黑色背景。

```
setApplicationBackGround(400, 400, false, 0x000000);
```

## 创建计分板



scoreBoard 变量在 GameFrameWork.as 类文件中定义并在 Main.as 中初始化。为了建立 scoreBoard，我们需要给 scoreBoard 的字体建立一个字体格式(scoreBoardTextFormat)，然后在创建 scoreBoard 所需的所有文本元素。在 stub 游戏中我们只需要一个元素就可以了。我们传进 SCORE\_BOARD\_CLICKS 常量，这个常量在 Main.as 类文件中的变量定义部分中定义的，在 public scoreBoard.createTextElement 函数中被传入的常量就会为计分板创建一个 SideBySideScoreElement 类实例。

```
scoreBoard = new ScoreBoard();  
addChild(scoreBoard);  
  
scoreBoardTextFormat= new TextFormat("_sans", "11", "0xffffffff", "true");  
  
scoreBoard.createTextElement(SCORE_BOARD_CLICKS, new SideBySideScoreElement(25, 5,  
15, "Clicks", scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));
```

在这里，我们设置“Clicks”字符串作为标签，并将其放置在计分板的 25, 5 坐标处。我们设置“Clicks”标签与内容文本之间的空隙为 15 像素。标签的宽度是 25 像素。我们在之前的 scoreBoardTextFormat 中已经设置好了标签和内容文本的字体格式。

## 创建标题画面

所有的 BasicScreen 实例都用一个相似的方式设置。都在 GameFrameWork.as 类文件中定义并在 Main.as 中的 init 函数里初始化。首先我们要创建一个独立的 TextFormat 对象(screenTextFormat)在所有的画面中共用。实际上我们可以让每个画面的字体格式都不一样，但是为了简化我们的示例游戏，我们就统一成一种格式好了。我们还要给个 BasicScreen 实例中的 SimpleBlitButton 实例创建一个字体格



式。

```
screenTextFormat = new TextFormat("_sans", "16", "0xffffffff", "false");  
  
screenTextFormat.align = flash.text.TextFormatAlign.CENTER;  
  
screenButtonFormat = new TextFormat("_sans", "12", "0x000000", □  
  
"false");  
  
titleScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE,□  
  
400,400,false,0x0000dd );  
  
titleScreen.createOkButton("OK", new Point(170, 250), 40, 20, □  
  
screenButtonFormat, 0x000000, 0xff0000,2);  
  
titleScreen.createDisplayText("Stub Game", 100,new Point□  
  
(145,150), screenTextFormat);
```

titleScreen对象通过BasicScreen实例的id值作为传入的常量来创建。我们用FrameWorkStates常量作为BasicScreen的id值。titleScreen的参数为宽度(400)，高度(400)，是否透明(false)，背景颜色(0x0000dd)。

下一步我们必须确定是不是要创建文本或者OK按钮在画面上。这些都通过调用BasicScreen类的public函数来设置。titleScreen.createOkButton函数创建了一个带有OK文本的SimpleBlitButton按钮实例，坐标为 170-250，宽度是 20.还传递了一个screenButtonFormat作为OK文本的字体格式，OFF状态颜色为黑色，OVER状态颜色为红色。最后我们传进了一个数字，2，作为文字位移的数值。

createDisplayText函数把文字放在titleScreen上，文字是Stub Game，宽度是 100，坐标是



145-150 , screenTextFormat则是文本的字体格式。

## 创建介绍画面

instructionsScreen 的初始化与 titleScreen 挺相似的。一些不同的地方就是用 FrameworkStates.STATE\_SYSTEM\_INSTRUCTIONS作为BasicScreen实例的id值，按钮的新文本是 Play ,画面上的新文字是Click the mouse\n10 times。你可能注意到一些尺寸和坐标也有些不一样。“\n”的意思是“强制换行”。当它加入到一个字符串里的时候，文本域就会被迫开启一个新行，之后的文本就会在新行里显示出来。

```
instructionsScreen = new BasicScreen(FrameworkStates.STATE_SYSTEM_INSTRUCTIONS,400,400, false,0x0000dd);

instructionsScreen.createOkButton("Play", new Point(150, 250), 80, 20,screenButtonFormat, 0x000000, 0xff0000,2);

instructionsScreen.createDisplayText("Click the mouse\n10 times",150,new Point(120,150), screenTextFormat);
```

## 创建游戏结束画面

gameOverScreen 跟 titleScreen 的初始化也挺相似，不同的就是用 FrameworkStates.STATE\_SYSTEM\_GAME\_OVER作为BasicScreen实例的id值，按钮的文本是OK，新的文字就是Game Over。你还会发现一些尺寸和坐标也不一样。

```
gameOverScreen = new BasicScreen(FrameworkStates.STATE_SYSTEM_GAME_OVER,400,400,false,0x0000dd);
```



```
gameOverScreen.createOkButton("OK", new Point(170, 250), 40, 20,screenButtonFormat,  
0x000000, 0xff0000,2);
```

```
gameOverScreen.createDisplayText("Game Over",100,new Point(140,150),screenTextFormat);
```

## 创建关卡画面

levelInScreen不同于其他三个画面的原因是它不用BasicScreen类的OK按钮。它利用名为levelInScreen自定义Main.as变量。这个变量保存了要加入levelInScreen对象的新关卡默认文字。

```
levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN, 400, 400, true, 0xaaff0000);  
  
levelInText = "Level ";  
  
levelInScreen.createDisplayText(levelInText,100,new Point(150,150),  
screenTextFormat);  
  
//Set standard wait time between levels  
  
waitTime= 30;
```

我们设置waitTime变量是 30( 如果帧频是 30 的话就相当于 1 秒 )。它用的是STATE\_SYSTEM\_WAIT中的内容作为默认等待时间。levelInScreen使用这个状态作为新关卡开始之前的过渡时间。

## 设置状态机的首状态

通过调用switchSystemState类的switchSystemState函数来开始运行状态机，你可以传进去你想要游戏首先进行的状态。在这个游戏里，我们要让状态起始于标题画面。

```
switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
```



## 开启游戏循环

通过设置frameRate变量和调用GameFrameWork类的startTimer函数后，框架就会开始游戏流程。

```
//create timer and run it one time
```

```
frameRate = 30;
```

```
startTimer();
```

## StubGame.as类

stub Game类是一个非常简单的游戏，只要玩家点击鼠标十次游戏就结束了。它用来展示Game.as与Main.as之间通过事件来通信的基础。你要把StubGame.as类文件保存在你使用的开发环境的工程文件夹中。

The Flash IDE' s file structure looks like this:

Flash IDE的文件结构是这样的：

```
[source]
```

```
    [projects]
```

```
        [stubgame]
```

```
            [flashIDE]
```

```
                [com]
```

```
                    [efg]
```

```
                        [games]
```

```
                            [stubgame]
```



## StubGame.as

Flex SDK(还有Flash Develop)的结构则是这样：

[source]

[projects]

[stubgame]

[flexSDK]

[bin]

[obj]

[lib]

[src]

[com]

[efg]

[games]

[stubgame]

## StubGame.as

下面是这个类的全部代码：

```
package com.efg.games.stubgame
```

```
{
```

```
// Import necessary classes from the flash libraries
```

```
import flash.display.Sprite;
```



```
import flash.events.MouseEvent;

import flash.events.Event;

import com.efg.framework.Game;

import com.efg.framework.CustomEventLevelScreenUpdate;

import com.efg.framework.CustomEventScoreBoardUpdate;

/**
 * ...
 * @author Jeff Fulton
 */

public class StubGame extends Game {

    //Create constants for simple custom events

    public static const GAME_OVER:String = "game over";

    public static const NEW_LEVEL:String = "new level";

    private var clicks:int = 0;

    private var gameLevel:int = 1;

    private var gameOver:Boolean = false;

    public function StubGame() {}

    override public function newGame():void {

        clicks = 0;

        gameOver = false;

        stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDownEvent);
```



```
dispatchEvent(new CustomEventScoreBoardUpdate(
    CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_CLICKS,String(clicks)));
}

override public function newLevel():void {
    dispatchEvent(new CustomEventLevelScreenUpdate(
        CustomEventLevelScreenUpdate.UPDATE_TEXT,
        String(gameLevel)));
}

override public function runGame():void {
    if (clicks >= 10) {
        gameOver = true;
    }

    checkforEndGame();
}

public function onMouseDownEvent(e:MouseEvent):void {
    clicks++;

    dispatchEvent(new CustomEventScoreBoardUpdate(
        CustomEventScoreBoardUpdate.UPDATE_TEXT,
        Main.SCORE_BOARD_CLICKS,String(clicks)));

    trace("mouse click number:" + clicks);
}
```



```
}  
  
private function checkforEndGame():void {  
  
    if (gameOver) {  
  
        dispatchEvent(new Event(GAME_OVER));  
  
    }  
  
}  
  
}  
  
}
```

## 我们要在这个类中做什么

这是我们创建的最简单的游戏了。当玩家点击鼠标，我们设置的MOUSE\_DOWN事件监听器就会增加clicks变量并给Main类派遣CustomEventScoreBoardUpdate事件来更新scoreBoard。runGame函数在每帧都被调用。如果clicks变量等于 10，gameOver变量就会被设置为true。runGame函数也会在每帧调用checkForEndGame函数。如果gameOver变量为true了，那么GAME\_OVER事件就会派遣给Main.as。

## 测试！

确保你所有的类文件都按照开发环境被放置在正确的文件夹中了。选择建立工程，输出影片，发布，或者其他开发环境中类似的选项。

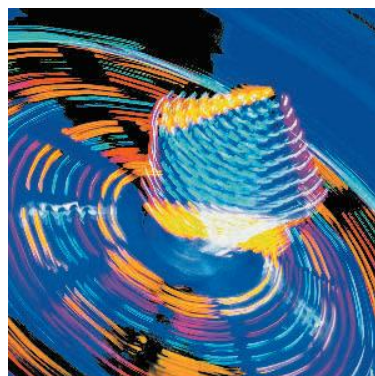
如果游戏运行不正确，最可能的问题就是类的路径错误。如果你在建立游戏的过程中有任何问题，重新检查一下框架的包结构路径是否正确。



## 总结

在这一章中，我们涵盖了许多本书中所有其余游戏的基础。我们创建了本书所有游戏都会使用的基本框架和包结构。还创建了一个非常简单的游戏来展示了框架里多种类之间的相互作用。

在下一章，我们会制作第一个完全体的框架游戏。Super Click（超级点击）是一个相对简单游戏，但是通过设计和代码编写有助于我们加深本章学习的框架概念。



## 第三章

# 创建超级点击

---

我们花了大量的篇章用在了创建一个游戏框架，但到目前为止，我们没有给他用武之地呢，接下来我们将创建一个简单的游戏，来试下它的身手。

让我们专注于我们的第一个游戏。“超级点击”，这个简单的游戏类似于早期的“Flash viral games”，当然它不是原创的，我们在这里只是利用它来说明这个游戏和主类是如何交互的。首先我们将在技术需求说明书中阐述在这个游戏中需要的元素，接下来，我们将讨论如何将他们作为游戏的各个部件整合在一起。在我们开始写代码值钱，我们将列出一些简单的技术设计文档。这个文档用于为SuperClick游戏创建一个main.as主类，这个类扩展子GameFrameWork类。它也用于创建SuperClick.as类。这个类扩展自Game框架类。

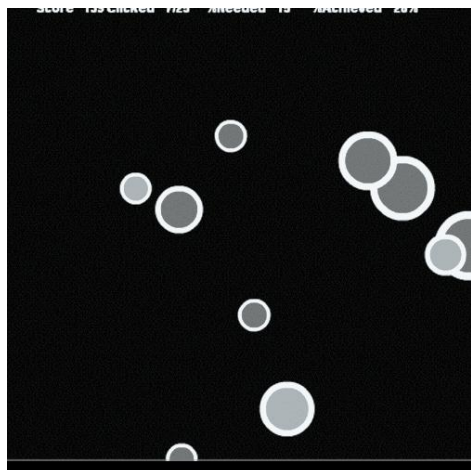


图 3-1. Super Click 游戏屏幕截图

## 创建 Super Click 游戏设计说明

我们接下来创建一个简单的游戏技术说明文档来描述基本的游戏玩法。但是更重要的是，它将描述我们将如何实现游戏的逻辑。为了好玩立刻投入到游戏的编码阶段其实也没有什么错的，实践能带给你更大的回报，但是当你开始设计用于商业或者受人委托而开发的游戏时，你最好还是在投入编码之前一定程度上规划好你的游戏逻辑。

## 定义游戏的基本要素

游戏基本要素描述游戏的设计意图和想法。游戏基本要素就是要描述游戏的设计意图和想法。

□ □ **游戏名:** Super Click ( 超级点击 )

□ □ **游戏目的:** 点中指定的圆，避开其他的圆。

□ □ **游戏玩法说明:** 蓝色和红色的圆将出现在屏幕上，从小慢慢的变大，当到达指定的大小后就会消失，玩家必须避开红色的圆点击VSE的圆，绿色的圆越早被点击，玩家得到的分数就越高。如果玩家没有点中的圆没有达到这一关要求的数量，游戏就结束。当玩家颠倒了红色的圆，游戏也会结束。



## 计算关卡数

级数计算描述了每一个级别的难度

□ □Number of blue circles per level**每一个级别中的蓝圆的数量**: 级别数的25倍。

□ □**错误的圆出现的百分比机会**: Level number plus 9 (25级以后百分比固定为40, 否则游戏将变的不可玩了)

□ □**圆的变大速度**: .01 乘以级数。

□ □**最大圆尺寸**: 5 减去级别数, 最小值为 1

□ □**进入下一个等级所需要的成功点击率**: 5乘以级别数加上10。

□ □**屏幕上出现的圆的最大数量**: 10 乘以级别数。

## 定义需要的基本界面

屏幕对象基本都是在第二章中介绍到的BasicScreen类的实例。你可能还记得, BasicScreen类允许类似 ok 按钮那样的按钮呈现在屏幕上, 这个按钮是BasicBlitButton类的实例。屏幕在开始新的一关卡的时候并不需要显示按钮, 所以屏幕上会有以下对象:

□ □**标签**: 包含 OK 按钮 和 Super Click 文字

□ □**介绍**: 包含 OK 按钮 和 Quickly Click Blue Circles 快速点击蓝色圆圈文字

□ □**游戏结束**: 包含 OK 按钮 和 Game Over 游戏结束 文字

□ □**级别**: 包含了级别文本 和级别变量不需要显示OK按钮

## 定义游戏变量

游戏变量用于空值游戏的逻辑和难易程度,

□ □circles: 这个数组变量保存了所有当前屏幕上出现的Circle实例。我们将创建自定义的Circle类,



并将类的实例对象保存在数组中。

□ `GameOver`: 如果玩家点到了红色的圆，或者没有点够这个等级要求的圆的数量，这个布尔变量设置为true

□ `level`: 保存了当前的游戏级别。

□ `numClicks`: 保存了玩家当前点中的圆的数量。

□ `numCreated`: 当前等级中已经创建爱你的圆的数量，直到 `maxCirclesOnScreen`(屏幕上出现的圆的最大数量)。

□ `maxScore`: 一个蓝色圆的最大分值。The maximum score decreases as the circles' size increases. They have an inverse relationship. The smaller the circle, the more points the player will score by clicking on it.

圆的尺寸变大分值就会减小，它们是呈反比的。玩家点到的圆越小，就会得到越高的分。

□ `percent`: 这个变量保存了点中蓝圆的百分比，计算方式如下： $100 * (\text{clicked} / \text{numCircles})$ 。

□ `score`: 保存了玩家的得分。

□ `tempCircle`: 对圆对象的引用,这个变量在circles数组中遍历中用到。

□ `scoreTexts`: 分数文本变量是一个数组类型的变量，保存了当前屏幕上出现的分数文本信息。我们将创建一个叫ScoreTextField 的类。将这个类的实例都存在scoreTexts这个变量中。

□ `tempScoreText`: 对ScoreTextField实例的引用，这个变量用于scoreTexts数组变量中遍历过程中。□ `textFormat`: 文本样式，最终这个变量会是一个TextFormat类型的变量用作格式化scoreTextField实例中的文本、

class that will be used to format the text for the ScoreTextField instances.

下面这些变量用于控制游戏的难度:



□ `circleGrowSpeed`: 圆的增大速度。

□ `circleMaxSize`: 每一个级别中圆的最大尺寸, 从5变到1

□ `maxCirclesOnscreen`: 一次在屏幕上出现的圆的数目 (所以在一开始的几帧内不要一下次都出现在屏幕上)

□ `numCircles`: 每一个级别蓝色圆的个数

□ `percentNeeded`: 每一个级别需要点中的蓝色圆的百分比

□ `percentBadCircles`: 当创建新的圆时。在一帧内出现红色圆的概率

## 定义自定义 ScoreBoard 记分牌元素

`ScoreBoard`类在第二章中已经创建了。这是一个非常简单的类, 它可以开发者将

`SideBySideScoreElement`类的实例放在它的显示列表中, 这个在第二章也有介绍。记分牌需要下面这些基本元素:

□ `SCORE_BOARD_SCORE`: 在`Main.as` 中的这个常量用来显示积分

□ `scoreLabel`: 指向分数标签对象

□ `scoreText`: 显示当前分数

□ `SCORE_BOARD_CLICKED`: 这个常量是用来显示最大点击次数在文本框中用 "/" 分开, 左边显示的是已点击的次数。

□ `SCORE_BOARD_PERCENT_NEEDED`: 这个常量是用来显示需要的被点击的百分比

□ `percentNeededLabel`: 显示 % needed

□ `percentNeededText`: 显示每一个级别需要点中的蓝色圆的百分比

□ `SCORE_BOARD_PERCENT_ACHIEVED`: 这个常量用来显示已经点击的百分比



□ `percentAchievedLabel`: 显示 % achieved 达成比例

□ `percentAchievedText`: 用来显示蓝色圆圈被点击的百分比

## 定义游戏流程

整个游戏分成若干个难度的关卡。每一关一开始在屏幕上随机的放置圆，直到放上去的圆的数量达到最大限制值(`circle.length == maxCirclesOnscreen`)。当创建圆时，`percentBadCircles`变量将用来检测创建哪一种圆(蓝色还是红色?)。圆会以`circleGrowSpeed`速度增长。圆每一帧都会增大它的`scaleX`和`scaleY`值。玩家点中越小的圆得到的分数就会越高，当玩家点中圆或者当圆的大小达到限定的最大值时，圆就会在离开屏幕，如果此时已经创建的圆小于圆数量的最大值，将会创建一个圆来代替刚刚的圆(`numCreated < numCircles`)。当点中蓝色的圆的时候，这个圆就会淡出，包含了玩家获得的分数的文本就会显示在刚击中的圆的附近。

## 判断关卡的结束

当所有的圆都出现在屏幕上时，关卡结束。当当前关卡结束前你务必要检查在`circles`数组变量中已经没有圆实例了。

如果`circles`数组的长度等于0，并且到目前为止创建的圆的数量等于这个级别需要的数量的时候，这关就过了。

## 判断游戏的结束

当玩家点中了红色的圆或者一关结束时还没有点中足够的蓝色的圆，游戏结束。(也就是点中的概率比`percentNeeded` (需要点中的概率)小)。



## 定义需要用的的事件

事件用于在框架中调度游戏状态的改变，更新计分板实例和关卡界面。我们将事件分为两类。第一类是简单的自定义事件。简单的由标准的dispatchEvent函数传递一些简单的字符串常量，他们不可以用来传递数据到侦听器函数。第二类事件是复杂的自定义事件，它们使用第二章创建的自定义的事件类，可以将数据传递到侦听器函数

以下就是简单的自定义事件常量：

□ □GAME\_OVER: 发往主程序以将状态切换至STATE\_SYSYEM\_GAME\_OVER

□ □NEW\_LEVEL: 发往主程序以将状态切换至 STATE\_SYSTM\_NEW\_LEVEL

这个程序仅仅需要几个复杂的自定义事件。当主程序需要改变记分牌时，我们需要一些基本的事件。

CustomEventScoreBoardUpdate.UPDATE\_TEXT 事件将传递两个变量：Key和Value。我们也将

CustomEventScoreBoardUpdate.UPDATE\_TEXT到主程序，传递新的级别到再自定义事件对象中，

更新记分牌上的分数：

**Event:** CustomEventScoreBoardUpdate.UPDATE\_TEXT (事件)

**Key:** Main.SCORE\_BOARD\_SCORE (键)

**Value:** 0 for a new game or the current score during game play (值，一开始是0，开始后就是当前分数)

更新在记分牌上显示的击中的文本

**Event:** CustomEventScoreBoardUpdate.UPDATE\_TEXT

**Key:** Main.SCORE\_BOARD\_CLICKED (键)

**Value:** 游戏一开始是 0/0，开始后就是 (击中数目+ “/” +numCircles)。

更新记分牌的需要的百分比

**Event:** CustomEventScoreBoardUpdate.UPDATE\_TEXT

**Key:** Main.SCORE\_BOARD\_PERCENT\_NEEDED

**Value:** 0% for new game or percentNeeded during game play for new level

更新计分板上的完成百分比

**Event:** CustomEventScoreBoardUpdate.UPDATE\_TEXT

**Key:** Main.SCORE\_BOARD\_PERCENT\_ACHIEVED

**Value:** 0% for a new game or percentAchieved during game play for new level

更新关卡界面的当前关卡数

**Event:** CustomEventLevelScreenUpdate.UPDATE\_TEXT

**Key:** “level” + level 变量



## 创建超级点击

现在我们可以拿着我们的设计文档来实现我们的游戏了。首先我们创建Game.as类。

为Game类创建一个新的Game.as文件，为Circle类一个新的Circle.as文件。

## 项目开始

就像第二章的stub游戏一样，

## 在 Flash IDE 中创建超级点击游戏项目

下面是在Flash IDE中创建项目的基本步骤：

1. 打开Flash,我目前用的版本是CS3，不过如果使用CS4或者CS5也是一样的。
2. 在[source][projects][superclick][flashIDE]目录下创建一个superclick fla文件。
3. 在[source]projects][superclick][flashIDE]目录下创建一个包结构：

[com][efg][games][superclick]com文件夹中有efg文件夹，games文件夹又在efg中，superclick文件夹又在games中。

4. 将Flash的帧速率设置为30FPS,将高和宽都设为400。
5. 将文档类设置为 com.efg.games.superclick.Main。
6. 因为我们还没有创建这个文档类，所以你会看到一个警告，没关系接下来我们就会马上来创建。
7. 然后加入框架包，在发布设置中。选择Flash>Actionscript 3 设置。
8. 点击浏览路径按钮。找到我们在第二章中创建的源文件的目录。
9. 选择类目录然后点击选中按钮，现在com.efg.framework包就可以在我们的游戏中使用了。

## 在 Flash Develop 中创建超级点击游戏。

下面是在Flash Develop中工程的步骤：



1.在[source][projects][superclick]文件中创建一个文件夹命名为：flexSDK。(如果你还没有做)。

2.打开Flash Develop 创建一个新的工程。选择Flex 3 Project,工程名为：superclick。位置应该是在 /source/projects/superclick/flexSDK目录，包应该是：com.efg.games.superclick。不要用Flash Develop的自动创建功能，务必确保Create Folder For Project未被选中。

3.点击OK创建工程。

4.添加类路径到这个项目的框架中：点击Project > Properties > Classpaths菜单项目

5.点击 添加类路径按钮，找到早先创建的源文件目录，选中这个文件含有类的子目录。

6.点击OK按钮然后按应用按钮。

现在你已经创建好了在这个框架下的工程的基本结构了

对于 Flex Builder Flash Builder, 或者别的开发环境的用户请参照相关的文档来创建项目并设置默认的编译类。

## 类文件列表

类文件列表就是所有我们要完成这个游戏用到的类的清单，所有的这些文件你都都在

com.efg.games.superclick包中。

□ □Main.as (扩展自 GameFrameWork.as 框架类)

□ □SuperClick.as (扩展自 Game.as 框架类)

□ □Circle.as

□ □ScoreTextField.as

## 创建 Main.as 类

让我们先看看目前我们做了哪些了。在接下来的几页，创建你的第一个基于此框架的游戏，将为学习制作后面的游戏做好准备，在这章中，我们前面使用了两个不同的开发环境来创建项目，如果你还没有准备好，那应该练一练再继续。你得创建一个Main类并且保存在你最近创建的工程的目录下。



下面是用于Flash IDE的文件目录结构

```
[source]
[projects]
    [superclick]
        [flashIDE]
            [com]
                [efg]
                    [games]
                        [superclick]
                            Main.as
```

下面是用于Flex SDK的文件目录



```
[source]
[projects]
    [superclick]
        [flexSDK]
            [bin]
            [obj]
            [lib]
            [src]
                [com]
                    [efg]
                        [games]
                            [superclick]
                                Main.as
```

下面是完整的代码：

```
package com.efg.games.superclick
{
import flash.text.TextFormat;
import flash.text.TextFormatAlign;
import flash.geom.Point;
import com.efg.framework.FrameWorkStates;
import com.efg.framework.GameFrameWork;
import com.efg.framework.BasicScreen;
import com.efg.framework.ScoreBoard;
import com.efg.framework.SideBySideScoreElement;
public class Main extends GameFrameWork {
    //custom score board elements
    public static const SCORE_BOARD_SCORE:String = "score";
        public static const SCORE_BOARD_CLICKED:String = "clicked";
    public static const SCORE_BOARD_PERCENT_NEEDED:String = "percent needed";
    public static const SCORE_BOARD_PERCENT_ACHIEVED:String= "percent achieved";
    public function Main() {
        init();
    }
    // init() is used to set up all of the things that
    //we should only need to do one time
    override public function init():void {
        game = new SuperClick();
        setApplicationBackGround(400, 400, false, 0x000000);
        //add score board to the screen as the seconf layer
        scoreBoard = new ScoreBoard();
        addChild(scoreBoard);
        scoreBoardTextFormat= new TextFormat("_sans", "11", "0xffffffff", "true");
```



```

scoreBoard.createTextElement(SCORE_BOARD_SCORE, new SideBySideScoreElement (25, 5, 15,
"Score",scoreBoardTextFormat, 25, "0",scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_CLICKED, new SideBySideScoreElement(85, 5, 10,
"Clicked", scoreBoardTextFormat, 40, "0/0",scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PERCENT_NEEDED,new SideBySideScoreElement(170, 5,
10, "%Needed", scoreBoardTextFormat, 50, "0%",scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PERCENT_ACHIEVED, new SideBySideScoreElement(260,
5, 10, "%Achieved", scoreBoardTextFormat, 60, "0%",scoreBoardTextFormat));
//screen text initializations
screenTextFormat = new TextFormat("_sans", "16", "0xffffffff", "false");
screenTextFormat.align = flash.text.TextFormatAlign.CENTER;
screenButtonFormat = new TextFormat("_sans", "12", "0x000000", "false");
titleScreen = newBasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE, 400,400,false, 0x0000dd );
titleScreen.createOkButton("OK", new Point(170, 250), 40, 20, screenButtonFormat, 0x000000,
0xff0000, 2);
titleScreen.createDisplayText("Super Click", 100,new Point(145,150),screenTextFormat);
instructionsScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS,400,400,
false,0x0000dd);
instructionsScreen.createOkButton("Play", new Point (150, 250), 80, 20, screenButtonFormat,
0x000000, 0xff0000, 2);
instructionsScreen.createDisplayText("Click the blue\ncircles",150, new
Point(120,150),screenTextFormat);
gameOverScreen = new BasicScreen
(FrameWorkStates.STATE_SYSTEM_GAME_OVER,400,400,false,0x0000dd);
gameOverScreen.createOkButton("OK", new Point(170, 250),40, 20,screenButtonFormat, 0x000000,
0xff0000, 2);
gameOverScreen.createDisplayText("Game Over",100,new Point(140,150),screenTextFormat);
levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN, 400, 400, true,
0xaaff0000);
levelInText = "Level ";
levelInScreen.createDisplayText(levelInText,100,new Point(150,150),screenTextFormat);
//set initial game state
switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
waitTime= 40;
//create timer and run it one time
frameRate = 30;
startTimer();
}
}
}

```

在第二章中的stub游戏中，我们详细的讨论了Main.as中init()函数所作的改变。现在对于超级点击



游戏来说，只是做了一点小小的变化。如果你很熟悉了就可以略过下面这节，否则就继续看吧。

## 为 Main 类导入类和定义变量

除了Flash库中的类，我没还需要格式化文本，我们还要导入Point类以便我们可以传递含有屏幕位置值的Point实例

我们要定义的仅有的变量就是ScoreBoard常量。在ScoreBoard上，我们有4个元素，因此我们都为他们定义好一个常量名字

```
public static const SCORE_BOARD_SCORE:String = "score";  
public static const SCORE_BOARD_CLICKED:String = "clicked";  
public static const SCORE_BOARD_PERCENT_NEEDED:String = "percent needed";  
public static const SCORE_BOARD_PERCENT_ACHIEVED:String = "percent achieved";
```

注意Main是GameFrameWork的子类，它拥有父类的一切的公开的方法和属性，可以直接调用。我们需要覆盖init方法来创建属于这个游戏的屏幕和计分板。

在这里我们在构造函数中就直接调用init()方法，在12章中，我们将介绍使用预加载机制，在这里我们不使用了，我们就在构造函数中简单的调用下init()方法。Main类的核心方法就是覆盖了GameFraneWork类的init方法。在这个方法中，我们要为每个特定的游戏修改框架。理解这个方法的作用是非常重要的，因为这本书中的每个游戏都对这个方法做了修改。

## 创建我们的游戏实例。

game变量在FameFrameWork中定义，但是实在Main.as中实例化的。我们将在下一节中创建一个Game类的子类：SuperClick.as。

```
game= new SuperClick();
```

## 设置游戏背景

appBackGroundBitmapData变量在FameFrameWork.as文件中已经定义了，但是值是在Main.as中



设置的。可以通过调用setApplicationBackGround()方法来设置：

```
setApplicationBackGround(400, 400, false, 0x000000); //四个参数分别是 宽和高400像素，不透明度，背
```

景色

## 创建计分板

计分板在GameFrameWork.as中定义，但要在Mian.as中实例化，要设置scoreBoard，我们需要为scoreBoard文本的外观创建一个TextFormat实例，然后在计分板上创建每一个文本元素，，有下面这些元素。

```
scoreBoard = new ScoreBoard(); //创建记分牌
addChild(scoreBoard);
scoreBoardTextFormat = new TextFormat("_sans", "11", "0xffffffff", "true"); //设置一种文本格式
//创建4个文本元素。
scoreBoard.createTextElement(SCORE_BOARD_SCORE, new SideBySideScoreElement(25, 5, 15, "Score",
scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_CLICKED, new SideBySideScoreElement(85, 5, 10, "Clicked",
scoreBoardTextFormat, 40, "0/0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PERCENT_NEEDED, new SideBySideScoreElement(170, 5, 10,
"%Needed", scoreBoardTextFormat, 50, "0%", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PERCENT_ACHIEVED, new SideBySideScoreElement(260, 5, 10, "%Achieved",
scoreBoardTextFormat, 60, "0%", scoreBoardTextFormat));
```

## 创建标题界面

所有的BasicScreen都有点类似，在GameFrameWork中定义的都将在Main中的init()方法中实例化，第一件事情就是设置一个独立的TextFormat对象让所有的界面元素共用，当然我们是可以为每一界面创建一个不同的TextFormat对象，但是简单起见，我们在这个例子就只用一个了。我们还为加入到BasicScreen实例中的SimpleBlitButton对象创建了一个TextFormat。

```
screenTextFormat = new TextFormat("_sans", "16", "0xffffffff", "false");
screenTextFormat.align = flash.text.TextFormatAlign.CENTER;
screenButtonFormat = new TextFormat("_sans", "12", "0x000000", "false");
titleScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE, 400, 400, false, 0x0000dd);
titleScreen.createOkButton("OK", new Point(170, 250), 40, 20, screenButtonFormat, 0x000000, 0xff0000, 2);
```



```
titleScreen.createDisplayText("Super Click", 100, new Point(145, 150), screenTextFormat);
```

创建titleScreen对象，它是一个BasicScreen（界面）的实例，创建该实例时，需要传递几个参数，分别是1、id:int，在FrameWorkStates中定义的常量FrameWorkStates.STATE\_SYSTEM\_TITLE；2、width:Number，界面宽400；3 height:Number，界面高400；4 isTransparent:Boolean,界面是否透明，不透明；5 color:uint，界面背景色 蓝色。

接下来我们创建一个“OK”按钮。BasicScreen类给我们提供了两个公开方法：1、createOkButton 用来在界面内创建按钮。2、createDisplayText 用来在界面内创建文本对象。这里创建 OK按钮就用titleScreen.createOkButton方法了。这个方法有如下参数：1、text:String,按钮的标签；2、location:Point,按钮出现的位置；3、width:Number,按钮的宽。4、height:Number,按钮的高度；5、textFormat:TextFormat,按钮标签的文本格式；5、offColor:uint=0x000000,鼠标移开后按钮的背景色；6、overColor:uint=0xff0000,鼠标移上去时按钮背景的颜色。7、positionOffset:Number=0，按钮上文本的偏移值。

然后再用BasicScreen提供的createDisplayText方法在界面内创建一个文本对象：这个方法有如下参数：1、text:String,文本的内容；2、width:Number,文本条的宽度；3、location:Point,文本条的位置；4、textFormat:TextFormat文本的格式。

## 创建操作说明界面

创建操作说明界面instructionsScreen 跟上面的创建标题界面titleScreen很相似，只是传递进去的参数有些区别而已。其中有一个不同的是创建BasicScreen实例的时候穿进去的第一个参数换成了FrameWorkStates.STATE\_SYSTEM\_INSTRUCTIONS，按钮的文本变成了“Play”，文本条的内容变成了“Click the blue\ncircles”，其余就是一些尺寸上的变化。

```
instructionsScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS, 400, 400, false, 0x0000dd);  
instructionsScreen.createOkButton("Play", new Point(150, 250), 80, 20, screenButtonFormat, 0x000000,
```



```
0xff0000, 2);  
instructionsScreen.createDisplayText("Click the blue\ncircles", 150, new Point(120, 150),  
screenTextFormat);
```

## 创建游戏结束界面

创建游戏结束界面gameOverScreen跟上面的创建说明界面很像，只是换了几个文本、尺寸和位置。

```
gameOverScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER, 400, 400, false, 0x0000dd);  
gameOverScreen.createOkButton("OK", new Point(170, 250), 40, 20, screenButtonFormat, 0x000000,  
0xff0000, 2);  
gameOverScreen.createDisplayText("Game Over", 100, new Point(140, 150), screenTextFormat);
```

## 创建屏幕上的关卡界面

levelInScreen对象跟其他三个对象是不同的，因为它不需要用到按钮。在Main.as中定义了义变量：

levelInText，这个变量保存默认的前缀，用于添加到levelInScreen

对象的每一个新的级别文本框的前面。

```
levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN, 400, 400, true, 0xaa0000);  
levelInText = "Level ";  
levelInScreen.createDisplayText(levelInText, 100, new Point(150, 150), screenTextFormat);  
//Set standard wait time between levels  
waitTime= 30;
```

我们设置waitTime变量为30帧，相当于1秒的时间。这个变量就像STATE\_SYSTEM\_WAIT常量那样用作为默认的等待时间。当新的一关卡开始前有这么一段用于过渡的时间。

## 设置初始状态

Main.as一开始运行的时候由GameFrameWork类的switchSystemState方法来设置它的状态。

switchSystemState方法接收不同的状态常量来改变程序的状态。我们的这个游戏的开始界面是titleScreen标签界面。

```
switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
```



## 开始游戏计时

一旦我们设置好了帧速率并且调用了GameFrameWork中的starTimer方法后，整个框架就开始工作了。

```
//create timer and run it one time  
frameRate = 30;  
startTimer();
```

## 为超级点击创建 Game 类

超级点击的Game类的结构跟前面的stub游戏的基本结构类似。我们需要创建一个SuperClick.as类文件。并且保存在工程的目录下，如下面的文件结构所示：

这是基于Flash IDE 下的文件结构

```
[source]  
[projects]  
    [superclick]  
        [flashIDE]  
            [com]  
                [efg]  
                    [games]  
                        [superclick]  
                            SuperClick.as
```

这是基于Flex SDK下的文件结构（用于Flash Develop ）。

```
[source]  
[projects]  
    [superclick]  
        [flexSDK]  
            [bin]  
                [obj]  
                    [lib]  
                        [src]  
                            [com]  
                                [efg]  
                                    [games]  
                                        [superclick]  
                                            SuperClick.as
```



SuperClick.as 类占据了比较大的篇幅。下面我们将一步一步来介绍其中的代码。

## 为 SuperClick.as 导入必要的类和定义变量

导入所有的需要的框架的类和Flash库类。

```
package com.efg.games.superclick
{
    // Import necessary classes from the flash libraries
    import flash.display.Sprite;
    import flash.events.*;
    import flash.text.TextField;
    import flash.text.TextFormat;
    import com.efg.framework.Game;
    import com.efg.framework.CustomEventLevelScreenUpdate;
    import com.efg.framework.CustomEventScoreBoardUpdate;

    /**
     * ...
     * @author Jeff Fulton
     */
    public class SuperClick extends com.efg.framework.Game {
        //game logic and flow
        private var score:int;
        private var level:int;
        private var percent:Number;
        private var clicked:int;
        private var gameOver:Boolean;
        private var circles:Array;
        private var tempCircle:Circle;
        private var numCreated:int;

        //messaging
        private var scoreTexts:Array;
        private var tempScoreText:ScoreTextField;
        private var textFormat:TextFormat = new TextFormat("_sans", 12, "0xffff", "true");

        //game level difficulty
        private var maxScore:int = 50;
        private var numCircles:int;
        private var circleGrowSpeed:Number;
        private var circleMaxSize:Number;
        private var percentNeeded:Number;
        private var maxCirclesOnscreen:int;
        private var percentBadCircles:Number;
```



注意我们的类必须扩展子com.efg.framework.Game类。GAME\_OVER 和 NEW\_LEVEL两个常量已经在Game类中定义好了，我们可以直接使用了。我们也可以覆盖Game类中的方法，以确保Main类继承的方法能访问在SuperClick类中定义的属性。

我们将变量分成三个部分。第一部分我们管叫游戏逻辑和流程。这些变量用于确保游戏能按照它设计的方式来玩。

```
//game logic and flow
private var score:int;
private var level:int;
private var percent:Number;
private var clicked:int;
private var gameOver:Boolean;
private var circles:Array;
private var tempCircle:Circle;
private var numCreated:int;
```

score 分数变量 保存了玩家累积后的当前得分。level 关卡变量保存了玩家当前到达的关卡数。

percent 百分比变量保存了玩家在当前关卡中中点蓝色圆的百分比。gameOver 游戏结束变量一般都是false除非前面谈到的两个状况发生了。circles 圆数组变量保存了当前关卡内的所有Circle实例列表的引用。

Circle类将在下一节做介绍。tempCircle 临时圆变量保存了挡在circles中遍历时的遍历到的圆的引用。

numCreated 已经创建的圆变量保存了每一个关卡中已经创建的圆的数量。并不是所有的圆在同一时间都创建好的，这得取决于numCirclesOnScreen变量。

接下来，还有messaging消息变量，消息对象(ScoreTextField)保存在scoreTexts数组变量中，在玩家点中了蓝色的圆的时候，会根据玩家点击圆时圆的大小和游戏中设置的每个圆可得的最大分值( maxScore )来决定哪个消息对象该出现在游戏界面上。tempScoreText变量是对显示在游戏界面中的消息对象 ( ScoreTextField ) 的引用。textFormat变量为消息对象 ( ScoreTextField ) 指定了文本格式。

接下来就是些控制关卡难度的变量。这些变量在游戏技术说明中介绍。其中的一个变量有一个默认值。



这是maxScore 最大分值变量。

```
private var maxScore:int = 50;
```

一开始游戏的最大分值设为50。当玩家在蓝色圆还在最小的时候点中它时，可以得到的分数。当圆在屏幕上越变越大时，点击后得到的分值也会根据来调整变得越来越小。

## 为 SuperClick.as 定义构造函数和初始化方法

这个游戏的构造函数是空的，不需要任何的处理

```
public function SuperClick() {  
}
```

在以后的游戏中，我们也许会在构造函数中调用一些初始化函数来设置一些变量值或者一些必要的元素。超级点击是一个很简单游戏，这里就不需要做什么了。

```
override public function newGame():void {  
    trace("new game");  
    level = 0;  
    score = 0;  
    gameOver = false;  
    dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,  
Main.SCORE_BOARD_SCORE, "0"));  
    dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,  
Main.SCORE_BOARD_CLICKED, "0/0"));  
    dispatchEvent(new CustomEventScoreBoardUpdate (CustomEventScoreBoardUpdate.UPDATE_TEXT,  
Main.SCORE_BOARD_PERCENT_NEEDED, "0%"));  
    dispatchEvent(new CustomEventScoreBoardUpdate (CustomEventScoreBoardUpdate.UPDATE_TEXT,  
Main.SCORE_BOARD_PERCENT_ACHIEVED, "0%"));  
}
```

你可能已经关注到了在方法前面有个override 关键词。这是因为在Game.as中已经申明过此方法，在子类中我们覆盖掉此方法。

在这几行代码中我们将变量score分数设为0，level关卡数设为0，gameOver游戏结束变量设为false。差不多所有的游戏都要在一开始的时候设置这些变量。当然也有些游戏没有分数值，或者没有关卡数。但是几乎所有的游戏都会有游戏该结束的时刻。



在变量初始化好后，我们在Main.as中发出自定义的事件用以更新记分板上的值。传送出去的键是我们已经在Main.as中预先定义好了的。比如，更新记分板上的分数，我们就传递Main.SCORE\_BOARD\_SCORE键并且对应的值为0。

如果我们回顾一下第二章，在ScoreBoard类中调用ScoreBoard.update方法来更新的值其实跟用上面这个派遣事件来更新的值是同一个值。SuperClick不再直接调用这个函数了，而是发出一个事件被Main类监听到（scoreBoard是Main类内部声明的一个对象）。这样就可以在Main.as中操作scoreBoard记分板了。

解耦SuperClick和记分牌，并将Main作为控制器和沟通机制，是基本框架推荐的方式，你如果觉得更习惯于直接调用，也是可以的。但从Main中将ScoreBoard实例的引用传向Game类显得更加明智些。或者在Main中提供getter和setter方法来让Game或者他的子类来间接的更新记分板。

如果ScoreBoard记分板类已经在Game类中被设置了侦听更新事件（替换掉Main.as来接收信息）。ScoreBoard记分板就不需要对Game类有引用。这样做本质上是没有什么错的，但是，一旦

## 定义 newLevel 函数

当Main类处于STATE\_STEM\_NEWLEVEL状态时会调用newLevel方法、它的主要任务是为新关卡重置有关变量。

每一关蓝色圆圈的数量(25个乘以级别)

- Chance for a bad Circle: (level + 9 or 40% after level 25) 改变红圈出现的几率（当前关卡数+9，在25级以后设为40%）
- Circle growth speed (.01 \* level) 圆增大的速度(0.01\*当前关卡数)
- Maximum Circle size (5 \* level or 1 after level 5) 圆圈的最大尺寸（5乘以关卡数，关卡数大于5之后设为1）
- Percent successful Clicks needed to move to next level (10 + (5 \* level)) 进入下一级别的关卡数要达到的点中百分比（10+（5乘以当前关卡数））
- Maximum number of Circles on screen: (10 \* level) 界面上的圆可出现的最大数量
- This function also sends custom Event instances for the following: 这个函数也发出自定义事件：

- Sends custom events to Main reset ScoreBoard 发出自定义事件到Main以重置记分板
- Sends custom event to Main to update Level Screen text 发出自定义到Main更新关卡界面文本。

下面是newLevel方法的完整代码：

```
override public function newLevel():void {  
    trace("new level");  
    percent = 0;  
    clicked = 0;  
    circles = [];  
    scoreTexts = [];  
    level++;  
}
```



```
numCircles = level * 25;
circleGrowSpeed = .01*level;
circleMaxSize = (level < 5) ? 5-level : 1;
percentNeeded = 10 + (5 * level);
if (percentNeeded > 90) {
    percentNeeded = 90;
}
maxCirclesOnscreen = 10 * level;
numCreated = 0;
percentBadCircles = (level < 25) ? level + 9 : 40;
dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_PERCENT_NEEDED,
String(percentNeeded)));
dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_CLICKED, String(c
licked + "/" + numCircles)));
dispatchEvent(new CustomEventLevelScreenUpdate(CustomEventLevelScreenUpdate.UPDATE_TEXT,
String(level)));
}
```

一开始方法中重置了进入下一关卡所涉及到的变量。percent变量是通过当前关卡需要玩家击中蓝色圆圈的百分比，clicked变量是玩家在当前关卡点中蓝色圆的数量。circles数组变量是所有Circle对象的实例集合（蓝色的和邪恶的红色的圆都有）。每过一个关卡level变量就增加1（level++）。然后进入为新的关卡计算

circleMaxSize变量用于控制游戏的难度，在低级别的关卡中，圆能够很大甚至超过它自身最大尺寸的100%，所以玩起来简单。当玩家在第一到第四关，我们将圆的最导致设置为5减去当前关卡数。所以在第一关，圆的最大尺寸是4（或者叫400%）。这使得圆在屏幕上呆的时间更长，也就更容易点击。当玩家在圆越小的时候点中它你得到的的分值就会越高。

你可能对下面这个表达式比较陌生，里面有？和：。这个是三元操作符，用这个表达式可以简化语句。

简单介绍下：如果level<5就将circleMaxSize的值设为5减级别，如果 level>=5那么设为1。

```
circleMaxSize = (level < 5) ? 5-level : 1;
```

numCircles变量的值设为当前关卡数乘以25。在更高的关卡中，将会有横多的圆需要点击。这个游戏过了第三关就变得很难完成了，但是可以预料到每一关都会有几百个圆了。

圆变大的速度是 0.01乘以当前关卡数，所以在第一关，圆增大的很慢。你可以看到在自定义类Circle了解到，圆从它自身的50%开始慢慢变大。所以。在第一关圆增大到它的原来的大小需要用5帧的时间，因



为在第一关circleMaxSize值是4 ( 5 - 关卡数 = 4 ), 从开始变大到离开屏幕需要经历45帧。

percentNeeded的值一开始是10, 然后每过一关就加上(5\*当前关卡数)。这个值表示你要过这关就必须点中蓝圆的个数, 当这个值超过90的时候, 我们就将它设为90。

maxCirclesOnScreen变量的目的是使得界面上不要同时出现太多的圆圈。

percentBadCircles变量后面也是用了三元表达式 “?:”, 它的用途是在创建新的圆时, 出现红圆的概率。做多是40%。

最后, 发出自定义事件。首先发出事件来更新计分板上的需要击中圆的数量和更新已经击中的数量。

最后发出的事件你还没有见到过。它的作用是告诉关卡预进入界面更新界面上的文本。接下来关卡预进入界面就呈现出来, 然后新的关卡数也已经修改好并呈现了出来。

## 调用 runGame 方法。

Main类在游戏处于STATE\_SYSTEM\_GAMEPLAY状态时循环调用runGame方法。

```
override public function runGame():void {  
    trace("run game");  
    update();  
    checkCollisions();  
    render();  
    checkforEndLevel();  
    checkforEndGame();  
}
```

在游戏的每一帧, runGame中的这些函数都会按顺序的调用。update函数会修改在界面上出现的圆和分数文本的值。checkCollisions函数会检查玩家点击了哪个圆。render函数会更新那些没有被玩家点击的圆的缩放比。checkForEndLevel和checkForEndGame函数将通过判断游戏过程中的变量等有关数据来决定是否让Mian类改变当前的状态。这些函数将在接下来详细讲解。



## 定义 update 方法

超级点击在每一帧都会调用update方法，它的职责就是创建新的圆，更新当前界面上的圆的尺寸，移除那些已经达到最大尺寸的圆，它还负责更新ScoreTextField分数文本实例中的文本，如果它们在界面上呆的时间超过了他们的生命周期就移除它。

```
private function update():void {
    if (circles.length < maxCirclesOnscreen && numCreated < numCircles) {
        var newCircle:Circle;
        if (int(Math.random() * 100) <= percentBadCircles ) {
            newCircle=new Circle(Circle.CIRCLE_BAD)
        }else {
            newCircle=new Circle(Circle.CIRCLE_GOOD)
            numCreated ++;
        }
        addChild(newCircle);
        circles.push(newCircle);
    }

    // Checks circles every frame for size and adds
    //to their nextScale property
    // if nextScale is larger than the max, removes the circle
    var circleLength:int = circles.length-1;
    for (var counter:int = circleLength; counter >= 0; counter--) {
        tempCircle = circles[counter];
        tempCircle.update(circleGrowSpeed);
        if (tempCircle.nextScale > circleMaxSize || _
tempCircle.alpha <0 ) {
            removeCircle(counter);
        }
    }

    var scoreTextLength:int = scoreTexts.length-1;
    for (counter= scoreTextLength; counter >= 0; counter--) {
        tempScoreText = scoreTexts[counter];
        if (tempScoreText.update()){ //returns true is life is over
            removeScoreText(counter);
        }
    }
}
```



## 将圆添加到界面上

如果屏幕上的所有圆(包括红的和蓝的)的数量小于maxCirclesOnScreen变量值。并且已经创建的蓝色圆的数量(也就是numCreated值)小于当前关卡所允许创建的蓝色圆的数量(numCircles),那么就创建一个新的圆。

如果需要创建一个新的圆,那么首先构造一个0-99的随机数去跟percentBadCircles变量值去比较,如果小于等于这个变量就创建红色的圆,否则就创建蓝圆。

Circle的构造函数接收一个整型参数。这个参数的作用是确定需要创建的是红色的圆还是蓝色的圆。一般就用Circle类中已经定义好的两个静态常量作为参数:CIRCLE\_BAD 和 CIRCLE\_GOOD,前面这个参数用以创建一个红色的圆,后面的这个参数用于创建一个蓝色的圆,如果创建蓝色的圆就将numCreated变量增加1。tempCirlce变量指向新创建的圆,并加入到circles数组变量中,然后用addChild方法加入到显示列表中。

## 更新界面上的圆

不管有没有添加新的圆对象到界面上,update方法都会遍历在界面上出现的圆。并用圆的增长速度作为参数调用每一个圆的update方法。更新后,检查这个圆是否已经达到了最大尺寸,如果是的,就移除它。在稍后的章节我们还会介绍Circle类的。如果这个圆被移除了,我们就以遍历到的当前序号来调用removeCircle方法。

## 更新分数文本实例

当玩家点击蓝色的圆,分数文本实例就会放置在界面上,分数文本实例会持续显示分数文本一段时间,然后会被移除。具体是,遍历scoreTexts数组变量,将其中的分数文本全部更新,如果这个分数文本已经不在它的生命周期内了,我们就从界面上把它移除。



## 优化循环

在Circle圆对象集合中遍历时，我们用了2次循环优化。第一个优化是创建circleLength变量。当我们构造for循环时，发现circles数组长度会减小1。然后我们就在遍历数组的时候使用这个变量。这样做有两个理由：

首先我们不用在每次循环迭代中都计算重新数组的长度，这样可以节省处理器计算时间。在每一次迭代中我们都要从数组中移除一个元素，并且在下一次迭代中试着再次计算circles的长度时，会导致发生一个运行时错误，为什么呢？既然数组的长度是1比一开始的长度要小了，我们可能会错过数组中的最后一个元素。

另外一个优化时我们反向遍历数组，这样做的话，当我们从数组中分离出一个圆实例时，不会跳过接下来从数组中分离出来的圆实例。

举个例子，当我们正向遍历这个circles数组时访问到第十个要被移除的元素时，我们需要用数组的splice方法删除掉这个元素。当我们这样做时，第十一个元素就会前移成为第十个元素，随后后面所有的元素都会前移。那么，在下一个循环迭代时，我们将不再访问第十个元素，其实这时候第十个元素是先前的第十一个元素。这肯定不是你想发生的吧。

我们也使用了一些更多的优化手段。使用变量tempCircle，并将它指向circles数组中的元素，那么接下来我们要用到circles中的元素并对其属性进行操作时，就不需要再频繁的检索数组了。要知道其实circles[counter]这样的操作也是会消耗cpu资源的。如果我们可以尽量少的去读取数组，我们何不这样做呢。

最优优化！优化一个循环，将数组的长度一开始就赋值给一个变量，在循环时，用该变量代替Array.length属性。

如果你要遍历数组的同时又要删除数组中的元素，可以考虑反向遍历该数组。如果对数组中的元素需要调用的次数多于一次，请申明一个变量指向这个元素可以让代码执行更有效率。

## 定义 removeCircle 删除圆方法

removeCircle方法有一个整型参数，这个参数表示要从circles数组中删除的圆的索引。它的任务就是清理掉这个圆对象释放它占得资源。

```
private function removeCircle(counter:int):void {  
    tempCircle = circles[counter];  
    tempCircle.dispose();  
    removeChild(tempCircle);  
    circles.splice(counter, 1);  
}
```



我们在移除圆对象的方法中调用了三个方法。首先，dispose()是销毁这个对象，此函数我们将在后面讲Circle类中讨论。Circle类的dispose方法会将Circle内部的对象多设为null，并且出去涉及到该对象所有的事件侦听器。接下来的removeChild()方法是将圆对象从superClick类的显示列表中移除。最后一个方法是将该圆对象从circles数组中删除。

## 定义 removeScoreText（移除分数文本）方法

removeScoreText方法有一个整形参数，此方法类似于上面介绍的removeCircle方法。这个参数表示要从scoreTexts数组中删除的ScoreTextField对象的索引号。它的任务是清理掉这个ScoreTextField对象释放它占得资源。

```
private function removeScoreText(counter:int):void {  
    tempScoreText = scoreTexts[counter];  
    tempScoreText.dispose();  
    removeChild(tempScoreText);  
    scoreTexts.splice(counter, 1);  
}
```

此方法中的代码跟removeCircle方法类似，这里就不再赘述了。

## 定义 checkCollisions（检测触碰）方法

当SuperClick类处于SYSTEM\_STATE\_GAMEPLAY状态时，SuperClick类的方法runGame会每一帧都调用checkCollisions方法。实际上，它并不是真的去检测两个物体之间的碰撞，也不是鼠标指针与圆对象的碰撞，而是遍历circles数组，找出数组中的那些被点击过的圆。当圆被点击后，圆有个属性checked会设为true。

```
private function checkCollisions():void {  
    var circleLength:int = circles.length-1;  
    for (var counter:int = circleLength; counter >= 0; counter--){  
        tempCircle = circles[counter];  
        if (tempCircle.clicked && !tempCircle.fadingOut) {  
            tempCircle.fadingOut = true;  
            if (tempCircle.type==Circle.CIRCLE_GOOD && tempCircle.alpha==1) {
```



```

var scoreAdjust:Number = 1 / tempCircle.scaleX;
var scoreAdd:int=maxScore * scoreAdjust;
addToScore(scoreAdd);
tempScoreText = new ScoreTextField(String(scoreAdd), textFormat, tempCircle.x, tempCircle.y,
20);
scoreTexts.push(tempScoreText);
addChild(tempScoreText);
}else if (tempCircle.type==Circle.CIRCLE_BAD) {
    gameOver = true;
}
}
}
}
}

```

这个方法也同样使用了我们先前讲的循环优化。这个循环的任务就是检测circles数组中的所有的Circle对象的checked变量的值，如果为true,则就要移除它了，计算新的分值(如果是蓝色的圆)，然后销毁它。如果点中的是红色的圆，此函数将gameOver变量设为true。

稍后我们会介绍addToScore函数，此函数需要一个参数。这个参数是将圆的最大分值maxScore乘以圆的尺寸的倒数，也就是圆的尺寸越大那么分值就越小。

不像AS2中，AS3的可见对象的scaleX, scaleY和alpha属性的值范围不再是0-100，而是0-1。

## 圆圈慢慢消失

当圆被点钟了不会从界面上立即移除圆,首先圆将渐渐淡去,看不见了再删除。这个是靠将tempCircle.fadingOut变量设置为true来实现的。当一个圆被第一次被检测到点击了，它的fadingOut变量值为false，我们将它设置为true。

```

if (tempCircle.clicked && ! tempCircle.fadingOutCircle) {
tempCircle.fadingOut=true.

```

当圆实例的alpha值仍为1（最大值为1）的时候，就可以知道渐隐还没有开始。当我们设置圆实例的fadingOut属性为true的同时，还要增加分数值，并将这个增加值显示在界面上。然后我们检测这个圆是不是蓝色的，并且它的alpha透明度是不是还是1，这意味着我们还没有把它作为一点中的圆处理。

```

if (tempCircle.type==Circle.CIRCLE_GOOD && tempCircle.alpha==1) {

```



```
var scoreAdd:int=maxScore * scoreAdjust;
addToScore(scoreAdd);
tempScoreText = new ScoreTextField(String(scoreAdd), textFormat, tempCircle.x, tempCircle.y, 20);
scoreTexts.push(tempScoreText);
addChild(tempScoreText);
```

用传进来的分数值，以及先前创建好的文本样式创建一个新的分数文本字段对象，并且放置在点中的圆的坐标。20为此分数文本在界面上显示的帧数。当fadingOut属性为true时，circle对象的update函数就会开始递减圆的透明值属性，而不再增加它的大小。当透明度减小至0的时候，就将该圆从界面上移除。

## 定义 AddToScore 函数

当一个蓝色的圆被点击后checkCollisions就会调用addToScore方法，并且传过去一个参数。这个参数是圆的最大分值的乘以(1/xScale)的取整。所以点中圆时，圆越大传过去的分值就越小，反之亦然。

```
private function addToScore(scoreAdd:Number):void {
    score += scoreAdd;
    dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_SCORE, String(score
)));
    clicked++;
    percent = 100 * (clicked / numCircles);
    dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_PERCENT_ACHIEVED,
String(percent)));
    dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_CLICKED, String(c
licked + "/" + numCircles)));
}
```

一旦蓝色的圆被点中，addToScore函数马上被调用。

```
score += int(maxScore * val);
```

新的分数值计算好后，就发出一个自定义事件CustomEventScoreBoardUpdate.UPDATE\_TEXT，将Main.SCORE\_BOARD\_SCORE作为键，将分数作为值。接下来，计算出checked和percent的最新值，并且也发出事件更新计分板。跟更新分数值一个方式。



## 定义 render ( 渲染 ) 方法

render方法很简单，就是遍历所有的圆对象，将他们的scaleX和scaleY值重新赋值一下。

```
private function render():void {  
    var circleLength:int = circles.length-1;  
    for (var counter:int = circleLength; counter >= 0; counter--){  
        tempCircle = circles[counter];  
        tempCircle.scaleX = tempCircle.nextScale;  
        tempCircle.scaleY = tempCircle.nextScale;  
    }  
}
```

## 定义 checkForEndOfGame 方法

checkForEndGame方法就是检测gameOver的值是不是true，如果是的那么发出GAME\_OVER自定义事件。当Main处于SYSYEM\_STATE\_GAMEPLAY状态时，每一帧都会调用runGame方法，而runGame方法则会调用checkForEndGame()方法。

```
private function checkforEndGame():void {  
    if (gameOver) {  
        dispatchEvent(new Event(GAME_OVER));  
        cleanUp();  
    }  
}
```

checkForEndGame方法简单的检测一下gameOver变量是不是为true.如果是的。就调用cleanup方法。然后发出GAME\_OVER自定义事件。Main类会逮住这个事件，然后切换游戏的当期状态到SYSYEM\_STATE\_GAMEOVER。只有在玩家点中了红色的圆，或者checkForEndLevel方法检测到玩家玩完某一关时，percent值还比percentNeeded小的情况下gameOver变量值才会设置为true。

## 定义 checkForEndLevel 方法

checkForEndLevel方法处理关卡的结束状态：

1. 检测是不是界面上没有圆了(circles.length == 0)，创建的圆的数量(numCreated)是不是等于这关



所需要创建的数量。

2. 当两个条件都为true，那么就可以进入下一关了。

3. 基于上面的检测，如果不都是ture,那么将gameOver变量设为true，或者发出一个简单的自定义事件 (NEW\_LEVEL)。

下面是完整的checkForEndLevel方法的代码：

```
private function checkforEndLevel():void {  
    if (circles.length == 0 && numCreated == numCircles && scoreTexts.length == 0) {  
        if (percent >= percentNeeded) {  
            dispatchEvent(new Event(NEW_LEVEL));  
        } else {  
            gameOver = true;  
        }  
    }  
}
```

如果此关已经完了，但是玩家并没有点中足够多的蓝色圆圈，那么gameOver变量就会设为true。如果玩家点钟了祖国多的蓝色圆圈，那么NEW\_LEVEL自定义事件就会发出。Main类就会逮住这个事件，将状态切换至SYSYEM\_STATE\_NEWLEVEL。

## 定义 cleanUp 方法

cleanUp方法循环遍历circles数组中的Circle对象，移除并且销毁所有的对象。当游戏达到gameOver标准的时候checkForEndGame就会调用这份方法。它也会同时清理掉scoreTexts数组中的对象。

```
private function cleanUp():void {  
    var circleLength:int = circles.length-1;  
    for (var counter:int = circleLength; counter >= 0; counter--) {  
        removeCircle(counter);  
    }  
    var scoreTextLength:int = scoreTexts.length-1;  
    for (counter= scoreTextLength; counter >= 0; counter--) {  
        removeScoreText(counter);  
    }  
}  
}  
} // close class
```



```
} // close package
```

在最后一个方法结束后，还有两个“}”符号。第一个是类的结束符号，另一个是包的结束符号。

这就是SuperClick的game类。接下来我们来看最后的两个类。

首先出场的是Circle类：

## Circle 类

Circle类动态的创建了一个圆形。它是sprite类的子类。所以当它被点击时，可以发出

MouseEvent.CLICK事件。

## 定义 Circle 类

创建Circle类的两个静态常量CIRCLE\_GOOD 和 CIRCLE\_BAD。

以下是以下公开的属性：

□ `clicked`: 布尔值，是否被点击了。

□ `type`: 整型变量: CIRCLE\_GOOD 或者 CIRCLE\_BAD. 这个变量也可以设为布尔值类型，但是设为整型更有扩展性，比如以后我们可以增加一种类型的圆(CIRCLE\_POWER\_UP) 当玩家点钟后可以延长时间。

□ `nextScale`: 这个属性保存了接下来圆的scaleX和scaleY值。Circle类的内部方法update会更新此值。此变量在SuperClick对象的update方法中被赋值，在SuperClick对象的render方法中会调用Circle对象的update()方法。

□ `fadingOut`: 这是个布尔变量。当一个蓝色的圆被点中了，这个值就设为true.所以在创建Circle对象是必须附带参数指明了这个圆是蓝色的还是红色的。

## 在包中创建 Circle 类

跟Main.as文件和SuperClick.as文件匹配，Circle类也被放在了特定的包文件中。

下面是Flash IDE的文件结构：



---

/source/projects/superclick/flashIDE/com/efg/games/superclick/Circle.as

下面是Flex SDK的文件结构 (使用 Flash Develop):

/source/projects/superclick/flexSDK/src/com/efg/games/superclick/Circle.as

## 导入 Circle 类需要用的的类和申明有关变量

创建一个Circle对象时，我们需要以CIRCLE\_GOOD和CIRCLE\_BAD两个常量中的一个作为参数。

clicked变量设为false。fadingOut变量设为false。nextScale属性不许要设值，他在Game.update方法中赋值。

clicked变量在游戏逻辑的触碰检测部分用到。如果圆被点击了。这个属性设为ture。在SuperClick的checkCollisions方法中会检测此值。

```
package com.efg.games.superclick
{
    // Import necessary classes from the flash libraries
    import flash.display.Shape;
    import flash.display.Sprite
    import flash.events.MouseEvent;
    import flash.text.TextField;
    import flash.text.TextFormat;
    /**
     * ...
     * @author Jeff Fulton
     */
    public class Circle extends Sprite {
        //Constants used to define circle type
        public static const CIRCLE_GOOD:int = 0;
        public static const CIRCLE_BAD:int = 1;
        public var type:int;
        public var clicked:Boolean=false;
        public var fadingOut:Boolean = false;
        public var nextScale:Number;
```

## 定义 Circle.as 构造函数和初始化函数

构造函数有一个参数，这个参数的值可以CIRCLE\_GOOD和CIRCLE\_BAD两个静态常量之一。构造函数



内调用了init()方法。

```
public function Circle(typeval:int) {
    buttonMode = true;
    useHandCursor = true;
    init(typeval);
}

public function init(typeval:int):void {
    var shapeColor:Number;
    switch (typeval) {
        case CIRCLE_GOOD:
            //good circle
            shapeColor = 0x0000FF;
            type = typeval;
            break;
        case CIRCLE_BAD:
            //bad circle
            shapeColor = 0xFF0000;
            type = typeval;
            break;
    }
    graphics.clear();
    graphics.lineStyle(2, 0xffffffff);
    graphics.beginFill(shapeColor);
    graphics.drawCircle(5, 5, 8);
    graphics.endFill();
    x = int(Math.random() * 399);
    y = int(Math.random() * 399);
    scaleX = .5;
    scaleY = .5;
    nextScale = scaleX;
    addEventListener(MouseEvent.CLICK, clickedListener, false, 0, true);
}
```

Circle类的主要部分都在init()方法里了。注意buttonMode和useHandCursor属性都要设置为true。

这两个属性都是继承Sprite类得来的。

init()方法首先判断传进来的参数是CIRCLE\_BAD还是CIRCLE\_GOOD。我们在这里用switch语句，目的是当以后扩展游戏时候方便引入其他种类的圆圈。

我们用graphics属性画一个圆，并用先前设置好的颜色来填充。圆圈的线条样式为2像素宽白色。



接下来，我们用红色或者蓝色来填充圆。要按照这个顺序，先设置好线条样式lineStyle和填充颜色beginFill，然后绘图drawCircle，最后结束填充endFill。我们绘制了一个圆心在(5,5)半径是8的圆。

```
graphics.drawCircle(5, 5, 8);  
graphics.endFill();
```

动态绘图就这么简单，init方法中还剩下这些：

- 1.随机的为Circle对象确定一个位置。
- 2.将SscaleX和scaleY的初始化值设为0.5
- 3.将clicked属性设为false。
- 4.为点击创建侦听器。
- 5.创建时，将nextScale设为当前的缩放别

## 定义 Circle 类中的 update 方法

SuperClick对象会调用Circle对象的update方法。SuperClick对象的update方法会遍历界面上的所有的圆，并将growSpeed作为参数调用这些圆的update方法。

```
public function update(growSpeed:Number):void {  
    if (fadingOut) {  
        alpha -= .05;  
    } else {  
        nextScale += growSpeed;  
    }  
}
```

如果圆的fadingOut值为true，那么圆的alpha变量就要减小0.05。反之，圆的nextScale属性就增大growSpeed。

## 定义 Circle 类的销毁方法

销毁方法的作用是去掉那些不想要的对象，为Flash的垃圾收集机制服务。Game类的removeCircle方法会调用这个方法的。



```
public function dispose():void {
    removeEventListener(MouseEvent.CLICK, clickedListener);
}
```

注意，我们要确定移除所哟不必要的事件侦听器，这样才能确保不需要用的到的对象占用的内存彻底释放。

## 更新 Circle.as 的点击监听函数定义

当Circle实例被点击，MouseEvent.CLICK事件就发出去了。此事件的侦听函数就会将发出事件的Circle实例的clicked设为true。当SuperClick类的checkCollisions方法执行后，就会将checked属性true的Circle实例从界面清除。

```
private function clickedListener(e:MouseEvent):void {
    clicked = true;
    removeEventListener(MouseEvent.CLICK, clickedListener);
}

} //end class

} // end package
```

## ScoreTextField 分数文本类

分数文本对象类动态的创建一个TextField对象，这个对象包含了玩家点中的圆可获得的分数值。分数文本对象有它自己的生命周期，一旦时间到了就从界面上移除。

## 定义 ScoreTextField 分数文本类

为ScoreTextField定义一下私有属性。

3.       textField: 一个TextField类的实例用于显示分数值
4.       life: ScoreTextField对象被移除前存在的帧数也就是生命周期
5.       lifeCount: 相对于生命周期已经耗掉的帧数，此值每帧都更新，We also need the following constructor parameters:
  - i.       text: 显示的分数值文本
  - ii.       textFormat: 分数值文本的样式
  - iii.       x: 分数文本的x坐标
  - iv.       y: 分数文本的y坐标
  - v.       life: 分数文本的申明周期（单位：帧）



## 在包结构中创建 ScoreTextField 类。

跟Main.as文件和SuperClick.as文件匹配，ScoreTextField类也被放在了特定的包文件中。

下面是Flash IDE的文件结构：

```
/source/projects/superclick/flashIDE/com/efg/games/superclick/ ScoreTextField.as
```

下面是Flex SDK的文件结构 (使用 Flash Develop):

```
/source/projects/superclick/flexSDK/src/com/efg/games/superclick/ ScoreTextField.as
```

先来看看完整的ScoreTextField类代码：

```
package com.efg.games.superclick
{
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.display.Sprite;
    /**
     * ...
     * @author Jeff Fulton
     */
    public class ScoreTextField extends Sprite{
        private var textField:TextField = new TextField();
        private var life:int;
        private var lifeCount:int;
        public function ScoreTextField(text:String, textFormat:TextFormat, x:Number, y:Number, life:int) {
            this.x = x;
            this.y = y;
            this.life = life;
            this.lifeCount = 0;
            textField.defaultTextFormat = textFormat;
            textField.selectable = false;
            textField.text = text;
            addChild(textField);
        }
        public function update():Boolean {
            trace("scoreText update");
            lifeCount++;
            if (lifeCount > life) {
                return true;
            }else {
                return false;
            }
        }
    }
}
```



```
    }  
}  
public function dispose():void {  
    removeChild(textField);  
    textField = null;  
}  
}  
}
```

大部分的代码在前面将谈到的类中已经见过。私有属性textField,life,lifeCount。还有构造函数的参数(text, textFormat, x, y,和 life)先前也介绍过了。构造函数除了将传递进来的参数赋值给私有变量外，以传递进来的文本值创建了一个TextField对象，然后用addChild()方法将该对象加入到分数文本对象的显示列表中。

## 定义 ScoreTextField 对象的 update 方法

SuperClick.update方法会在遍历scoreTexts数组中的ScoreTextField对象时，调用所有ScoreTextField对象的update方法。每一次调用都会将ScoreTextField对象的lifeCount变量值加上1。如果加上1后变量值比生命周期大就返回true给SuperClick的update方法。此刻ScoreTextField对象也该被销毁了。

## 为 ScoreTextField 类定义 dispose 方法

此方法的作用在update方法中有介绍了。

## 测试

确保每一个类文件所在的文件夹跟前面介绍的一样。选择创建工程，导出影片或者发布，或者选择你的开发环境中类似的选项。

学习者犯得最多错误一般都是类路径设置问题。如果在编译过程中有错误，可以从检查框架包路径开



始。

## 总结

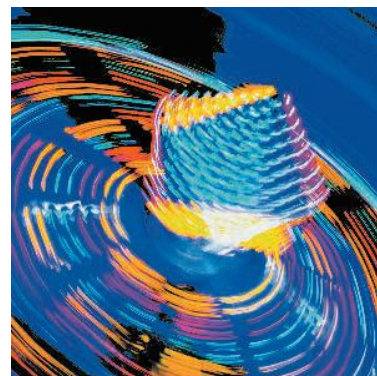
在这一章里，我们使用框架包最终完成了一个叫超级点击的全功能的游戏。我们创建了两个新类。

GameFrameWork类的子类Main类和Game类的子类SuperClick类。这两个类都是框架包中的类的子类。

另外我们还创建了两个与框架包无关的类：Circle类和ScoreTextField类。

在下一章节，我们将创建一个街机游戏叫做加农炮并且探索更多的游戏开发技巧，比如学习使用

BitmapData对象和精确的像素级碰撞检测。



## 第四章

# 御空加农炮的基础架构

在本书第一部分，学习了怎样创建一个游戏框架以及把这个框架应用到简单的游戏里。在本章，我们将会运用你在第一部分学习到的东西来做一个 AS3 游戏，包括：图形，声音，碰撞检测。现在，我们就把理论用到实践中去，开发 Flak Cannon 游戏。

本章，我们将注重支撑 Flak Cannon 游戏的必备基础。这包括设计游戏，创建素材，创建一个声音管理器，感受一下 Flex 和 Flash 两种库之间的不同，创建游戏难度和动画场景。下一章，我们就会把这些创造出来的东西构建为一个功能产品。





图[4-1] 加农炮

## 导弹指挥官的前情回顾

Atari 是第一个成功的视频游戏公司。它在 1972 年发布了 Pong 游戏做为开始，并在接下来的 30 年里面不断完善视频游戏（期间有换过不同的名字，包括 Midway West）。Atari 在上世纪 70 年代有好多很受欢迎的游戏，其中有 Gran Trak 10、Tank (under the Kee Games label)、Breakout、Football，还有在 1979 年，他们卖得最好的游戏-----Asteroids。1980 年，他们迫切需要后续行动，帮助他们保持在视频游戏行业的前列。

于是，Atari 在 1980 年发布了两个以军事武器为主题的游戏：Battlezone and Missile Command（战争地带和导弹指挥官）。Battlezone 是第一个 3D 射击游戏，Missile Command 则是截然不同的另一款游戏。这些游戏充斥了冷酷的恐惧的战争以至于在夜间新闻才被播出 and in the “drop and cover” drills of 1970s elementary school classrooms:核战大屠杀。这和那些 X 年代的孩子们在视频里感受到的恐惧是一样的，这也是这款游戏能够流行的一个因素。另一个因素就是玩这个游戏能够让你舒畅愉快。

Missile Command 模拟一场对 6 个城市的核弹攻击，而玩家则需要负责防御。玩家有 3 个独立的发射井，每个发射井有 10 枚反导弹导弹。利用准星来瞄准，玩家可以通过 3 个不同的火力按钮来发射导弹击落来袭的导弹，杀伤卫星，核轰来炸机。反导弹导弹不会直接去击毁来袭的敌人，而是飞去敌方导弹的目的地。当导弹达到预定目的地就会爆炸，摧毁范围内的任何敌方。

这个独特的游戏引导玩家接收各种有趣的策略，以此获取更高的分数。一些玩家在游戏开始会进行大规模的爆炸来尽可能的减少来袭的导弹，而另一些玩家则会采用“减少损失”的策略只保护一半的城市，这样就可以为剩下的幸存者保留更多的导弹。那三个不同的导弹基地也有一个有趣的挑战：训练 3 个手指去发射导弹，并且很精准的打击逼近的导弹。可以说，没有其他电子游戏或者一个射击游戏机制能够为玩家提供这样丰富的方式去部署防御。

这个游戏由 Dave Theurer and Rich Adam 设计，在 1980 年发布并取得了巨大的成功。虽然没有达到 Asteroids 的高度，但是还是有着卖了接近 20000 份的巨大冲击。游戏引起了经历了冷战的那一代人，甚至有可能是给这代人的潜意识中核战争担忧的一种释放。虽然在接下来的 30 年里面，这个游戏受到的关注越来越少，但是最近 NBC 播出了一段短片，围绕着 Missile Command 这个游戏及其游戏的间谍和暗号背景，证明这个游戏在流行文化历史中有着浓厚的一笔。

作为一个射击游戏，Missile Command 同时也是一个很有特色的游戏。奇怪的是，在这个游戏最为流行的时候，并没有很多的山寨它的游戏存在。然而，一些商业机构因为家庭游戏而开始发展起来了。Atlantis by Imagic for the Atari 2600（译者注：应该是个电影名）采取这个思想并使得电影里的人保卫了一个水底之城。Cinemaware 的电影 S.D.I 围绕 AtariST 和 Amiga 电脑之间的斗争来展开。在 1990 年代，Atari 尝试通过 Missile Command 在 Jaguar 视频游戏控制台的 3D 版本来重造当年的辉煌（但是失败了）。但是，除了这些（一些做了稍微更新的并发布在各种平台上的），几乎没有游戏是运用 Missile Command 的玩法风格。

在接下来的两章，我们将踏上征途，创建一个以 Missile Command 为基础的游戏。不是一个假冒商品。我们会借鉴 Missile Command 的一些元素，当然，也会添加我们自己的元素。



## 游戏设计

在我们的游戏中，你扮演一个在二战时期的海军军舰防卫者。你的任务就是在日本的敢死飞行队击中你的船队之前歼灭它们。下面是我们项目的简单设计：

- 游戏名称：Flak Cannon (发飞机)
- 游戏类型：射击类
- 游戏描述：玩家守卫舰队，防御来自飞行敢死队的袭击
- 玩家目标：不要让敌方飞机攻击到你的舰队并歼灭敌方
- 敌人描述：敌方飞机会从屏幕的上方或者旁边进入，并企图尽快飞到下方的舰队（也就是玩家守卫的舰队）。敌方的飞机就是一个炸弹，它会沿着直线飞行而且不会躲避玩家发射的炮弹。它们的目标就是炸毁玩家守卫的舰队。
- 敌人目标：摧毁玩家守卫的舰队
- 过关：当一波攻击中所有的飞机全部完成袭击的时候就会进入下一关。没用完的炸弹可以用来加分，舰队会补充完（if a player has earned them as a bonus）。
- 游戏结束条件：玩家要守卫的舰队全军覆没。
- 难度上升：游戏的难度上升时通过数学计算得出来的。敌军是按照一定的时间出现的，但是出现的位置和飞行路线是随机的。
- 奖励条件：游戏中有三个方式可以获得奖励。
  - 一次开炮打中多架飞机，奖励的分数就是击中的飞机数。
  - 击中从右往左飞过的“奖励飞机”，可以获得额外的 10 个弹药。
  - 当玩家获取到 10000 分的时候，军舰的数量会加一。

## 本章的游戏开发概念

而在第二部分的章节涵盖了使用同一个游戏框架来开发不同的游戏，当然，游戏开发主题会在这章介绍，在其他章节就不作介绍了。我们这样做的目的是为了最大限度地涵盖游戏主题和使游戏引擎的开发尽可能的多样化。在这一章我们将会粗略讲述一下游戏开发的相关概念。

6. 把素材添加到库里面
7. 处理 Flash，Flex 和 Flash Develop 库之间的差异
8. 创建一个基本的声音管理器
9. 创建一个难度设置



10. 在一个向量里面创建场景和动画

11. 创建动画场景

我们做的这个项目一开始会参考 Super Click，你可以在 flash IDE 或者 Flex 或者 flash develop 中创建这个游戏。在有需要的地方，我们会强调两个开发环境中代码的不同之处。这一章会集中在 flash IDE 中创建游戏。然而，我们写的代码都很容易转换到其他的开发工具中。这里是 FLA 文件的新属性：

- Package: com.efg.games.flackcannon
- Game Class: FlakCannon.as
- Document Class: Main.as
- Size: 600 \* 400
- FPS: 30
- Flex 素材位置: src/com/efg/games/flackcannon/assets

这时候，你可以在书的背后找一个网址来下载相关的资源。你可以在“/source/projects/ch4\_5\_flackcannon”这里找到相关的素材和代码。在这个目录里可以找到 Flash IDE 和 Flex SDK 的各个版本。

## 添加游戏素材到库

在 Super Click 中我们没有使用任何的 SWF 库素材。一切都是从 0 开始，慢慢编码。然而，这是一个很好的工作方式，根据你游戏中需要用到的素材去创建相关资源。对于 Flak Cannon，我们会用到图像和声音来制作游戏。

## 使用 GPL 来创建图像

如果每个玩游戏的人都知道编码和创建视图有多么困难，那多好啊。很多程序员觉得很难制作一些能够吸引住玩家一段足够长的时间来体验他们为玩家设计的一系列很酷的东西。

很庆幸的是，网络为开发人员提供了很好的资源，而且很多都是免费的。其中一个就是 Spritelib GPL，由 Ari Feldman 在 [www.flyingyogi.com/fun/spritelib.html](http://www.flyingyogi.com/fun/spritelib.html) 上面维护。这是一个免版税的图形库，这意味着你可以在用它来美化任何东西。本书中的好几个游戏都是使用这个图形库的，如果你有另外的预算，你可以使用另一个图形库：[www.istockphoto.com](http://www.istockphoto.com)。

很显然，在一个专业的工作室，你会有专业的设计艺术素材，所以如果你有这样的工作环境，你不需要自行创建任何的素材。事实上，很多工作室里面，设计师对程序员的艺术很无语，清楚你自己的工作环境，谨慎行事。在我们的游戏里面，我们会使用下面的图片，大多数都是来自 Spritelib GPL。

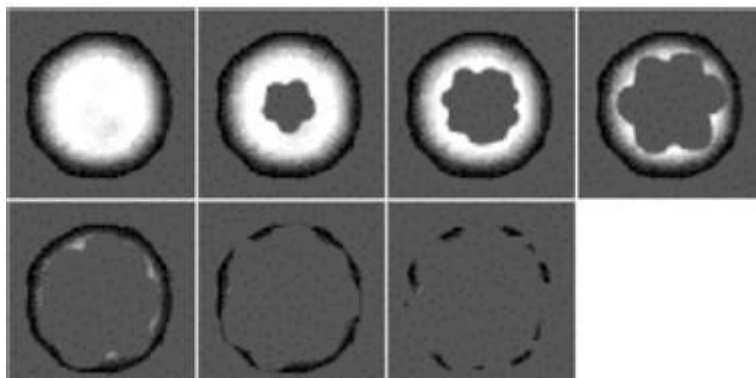
FLAK CANNON 的准备工作:

- Shot (库类名: ShotGif): 玩家射击敌方飞机的炮弹



图【4-2】玩家发射的炮弹

- Flak ( 库类名 : Exp1Gif . . Exp7Gif ) 高射炮弹药的爆炸就是玩家用来摧毁敌方飞机的，高射炮的弹药本身是没有伤害的，只有发射并爆炸之后才有用。这几个单一的图片会用来制成一个多帧的动画。



图【4-3】高射炮弹药爆炸动画

- Enemy plane ( 库类名 : PlaneGif ) 这就是主要的敌人，它直直地飞向玩家守卫的舰队。



图【4-4】飞机向下飞

- Enemy plane left ( 库类名 : PlaneLeftGif ) 飞机从左边飞出来，它在越靠近屏幕的下方飞进来，就越难被打下来。



图【4-5】飞机从左边飞进来

- Enemy plane right ( 库类名:PlaneRightGif) 飞机从右边飞出来，它在越靠近屏幕的下方飞进来，就越难被打下来。



图【4-6】飞机从右边飞进来



- Explosion ( 库类名 : Ex21Gif . . Ex25Gif ) 游戏中任何东西的爆炸动画。



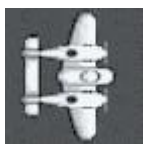
图【4-7】爆炸动画

- Ship ( 库类名 : ShipGif ) 这就是玩家要守卫的军舰，一开始玩家有 3 个军舰。



图【4-8】军舰

- Bonus plane ( 库类名 : PlaneBonusGif ) “奖励飞机” 总是从左往右飞，如果玩家击中它，就可以获得额外的 10 个弹药。



图【4-9】奖励飞机

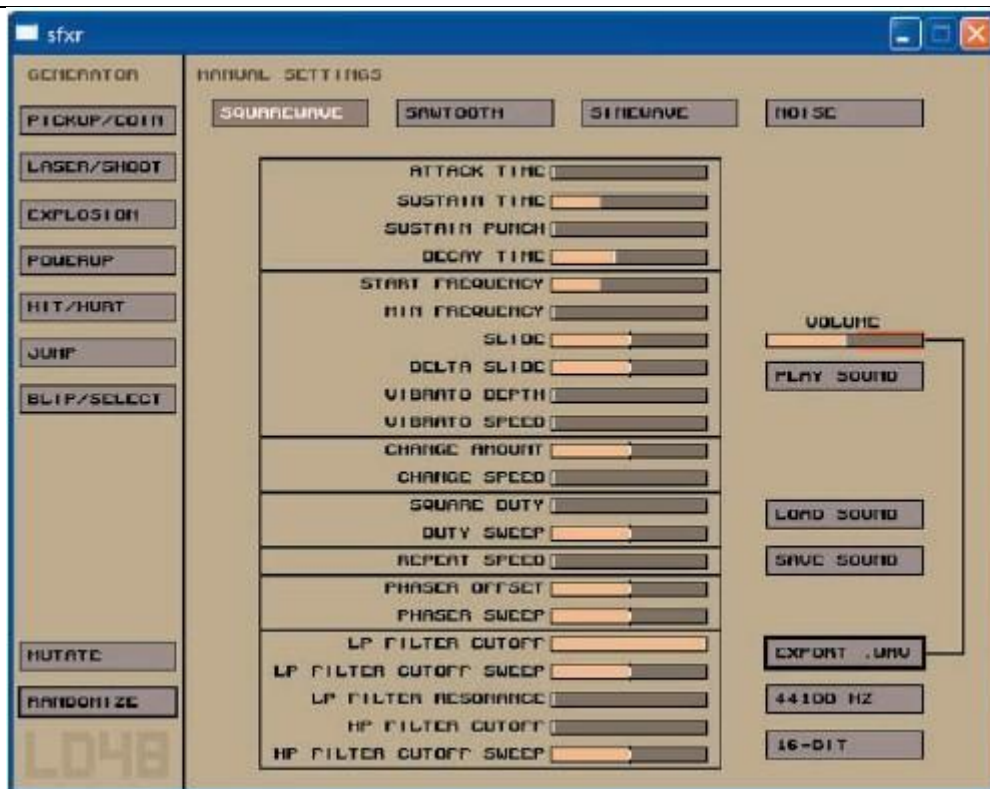
- Crosshairs ( 库类名 : CrosshairsGif ( 译者注 : 十字准星 ) ) 玩家通过鼠标来控制。弹药去到十字准星的中心就会爆炸造成杀伤。



图【4-10】十字准星

## 使用 SFXR 制作声音

找一个适合你游戏的声音估计比找图片还要难。很多的音效库和商业项目会严格控制你使用它们的声音，同时也可能很贵。在一个专业的环境或者工作室中，你可能会有专属的音效可以使用。当然你都可以购买一些音效库，例如这个：[www.soundtrangers.com](http://www.soundtrangers.com)。不过，就像使用图片一样，你可以利用那些很好的同时又免版权的音效资源。我们第一个体验到的音效资源是 SFXR，在图【4-11】可以看到。SFXR 是由 Tomas Pettersson 创建的，并且公布在 [www.cyd.liu.se/~tompe573/hp/project\\_sfxf.html](http://www.cyd.liu.se/~tompe573/hp/project_sfxf.html) 以供大家使用，有 PC 和 MAC 版本。



图【4-11】SFXR PC 使用界面

ludlum dare.com 博客评价说 SFXR 就像是“音效的 MS Paint”，SFXR 可以帮助你随机生成音效，并导出为 WAV 文件以供你游戏使用。创建一个音效之后，你可以自定义调整音效的相关属性。使用该程序相当简单，你点击左边的在 Generator 下面的一个按钮，就会随机生成一个音效。如果你喜欢这个音效，你可以点击 Export 按钮来导出它，如果你不喜欢，你可以再生成另一个音效，或者你可以操作屏幕中间的工具栏来改变音效。我用 SFXR 来创建 Flak Cannon 的音效（见表【4-1】），大概花了我 10 分钟来制作。即使你在一个专业的工作室里工作，它仍然是个宝贵的工具：通过它来创建一个音效，然后放在一个需要音效的地方，等到有专业的音效了，就替换上去。

表【4-1】Flak Cannon 的相关声音和对应的 Flash 类名：

类名	何时使用
SoundBonus	玩家击落“奖励飞机”并获得额外的弹药。
SoundBonusShip	玩家获得额外的军舰。
SoundExplodePlane	玩家击落一架敌军飞机。
SoundExplodeFlak	一个高射炮弹药爆炸。



---

SoundExplodeShip	一只军舰被毁.
SoundNoShots	玩家没弹药了.
SoundShoot	玩家投放炸弹.

---

## Flash and Flash Develop/Flex 中类库的不同

现在我们已经创建好图片和声音并把它们放到库里面去了。你需要根据游戏中需要的东西来正确地把他们关联起来。很幸运，库里的资源对于编程方法来说还是一样的，不过在对游戏中的类访问就不同了。当你正在把图像和声音嵌入到游戏中，Flash IDE 和 Flex 框架在实例化这些素材的时候会有一些不同。

这个游戏，我们已经用 Flash IDE 把所有需要用到的图片和声音资源都嵌入到 FLA 库中了。如果要在 Flex 中使用那些素材，我们必须发布成一个 SWF 文件（用一个新的名字），然后放到我们的 Flash Develop/Flex 项目中的 src 中的 assets 中(src/com/efg/games/flakcannon/assets).这样，我们就可以利用现有的素材库而不需要重新创建一模一样的素材文件夹。这个概念对于非 MP3 格式的声音文件来说是特别重要的，因为你不能直接把它们嵌入到项目中，例如，如果它们不是 SWF 文件库中的一部分，你不可以添加一个 WAV 文件到 Flex 应用程序中的。

你可能会问：“为什么 assets 文件夹放在 Flex 的源码中”。答案很简单。如果我们在根目录下放置 assets 文件夹，那么在关联素材的时候就要写成 ../../../../assets/flackassets.swf 这样，这样的话就比较繁琐了，把 assets 放在 src 那里，就显得简洁多了。如果这样给你带来了烦扰，你可以按照实际情况移动 assets 并修改相关代码。

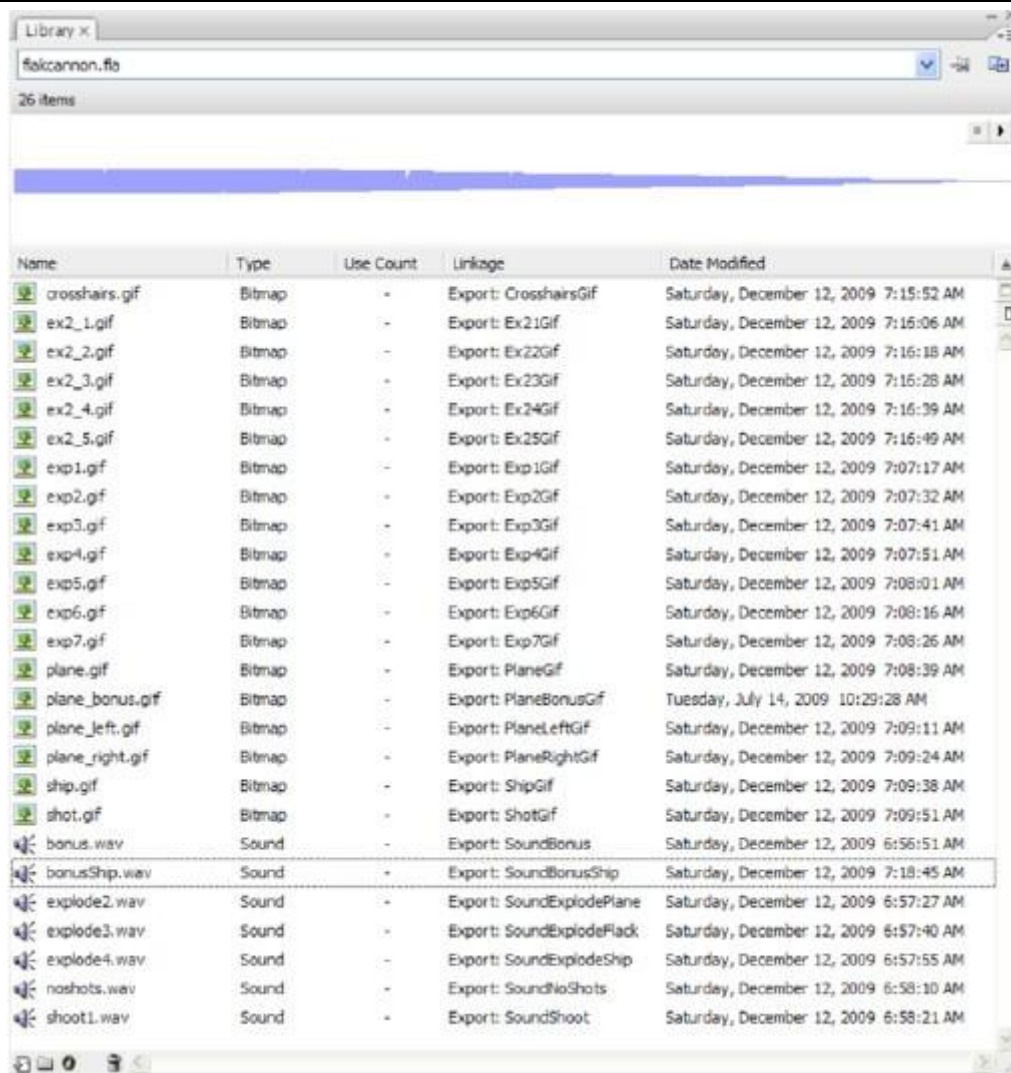
## 使用图形成素

我们先来看看 Flak Cannon 的图形成素是怎样嵌入到 Flex 和 Flash 中的。在 Game 类中，我们需要把库中的 crosshairs.gif 嵌入到游戏中。代码：

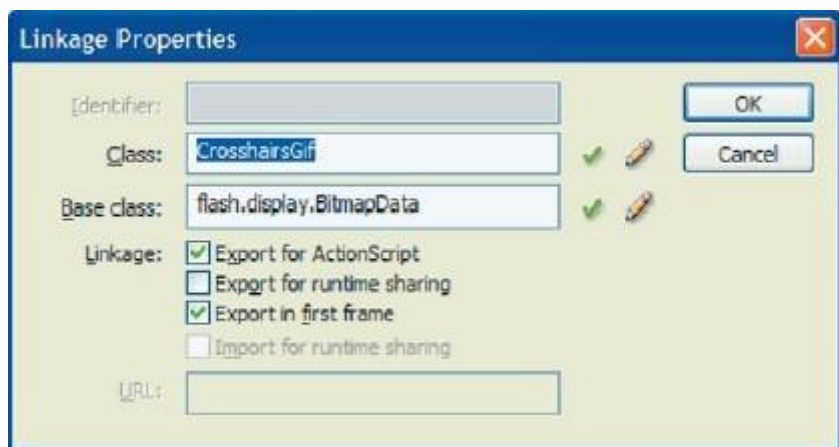
### /\*\*Flex Framework Only

```
[Embed(source = "assets/flakassets.swf", symbol="CrosshairsGif")]  
private var CrosshairsGif:Class;
```

根据你在 Embed 中的“symbol”中的名字，就是你在 Flash IDE 中引用到的名字。为库中的 Flash 素材创建一个关联（见图【4-12】），在库中找到这个素材，点击它，弹出选择菜单，然后选择关联（Linkage）操作。在 Class 输入框中写好类名（见图【4-13】），因为这是一个类，所以名字尽量要按照类名的风格去改。



图【4-12】FLA 库导出素材



图【4-13】导出 CrosshairsGif

当不在 Flash IDE 中开发的时候，运用图片的相关代码会有小小的改变。这是因为在 Flash IDE 中是用



BitmapData 来把所有的图片嵌入到库中的，而在 Flex 中则是通过 BitmapAsset ( BitmapClass 的子类 )。基于这个原因，如果你一开始在 Flash IDE 中编码，那你可能需要把 BitmapData 转化为 Bitmap。下面是 crosshairs.gif 图片的例子：

■ Flash 版本：

```
private var crosshairs:Bitmap;
...
crosshairs = new Bitmap(new CrosshairsGif(0,0));
```

■ Flex SDK 版本

```
private var crosshairs:Bitmap;
crosshairs = new CrosshairsGif();
```

在 Flash 版本中，我们希望 crosshairs 能够作为 Bitmap，这样我们就可以把它显示在屏幕上。因为在 Flash IDE 中图片是以 BitmapData 类型嵌入在 IDE 库中的，我们需要创建一个 crosshairs BitMap 实例并传给它一个调用来创建一个新的 CrosshairsGif 版本。

在 Flex 版本中，因为 CrosshairsGif 已经是 BitmapAsset ( Bitmap 的一个子类 ) 的类型嵌入在库中的，我们只需要简单的创建它的一个实例，然后就完成了。

关于 Flash/Flex 的素材嵌入的另一个问题就是在 Ship 类中了。我们希望变量 imageBitmapData 是 BitmapData 的实例而不是 Bitmap，下面是 Flex 中的嵌入版本：

```
[Embed(source = "assets/flakassets.swf", symbol="ShipGif")]
private var ShipGif:Class;
```

再一次注意：类创建使用了在 Flash IDE 版本中相同的关联名。在 Flash IDE 版本中，创建一个新的 BitmapData 实例，我们只需要简单的 new 关联名 ( 长，宽 )。

```
/****** Flash *****/
imagebd = new ShipGif(0,0);
```

对于 Flex，Bitmap 是 BitmapAsset 的一个属性，所以我们只需要稍稍修改一下调用就行了。我们不需要传递长度和宽度的参数，当创建 imageBitmapData 的时候，我们只需要简单地关联好 ship\_gif 的 BitmapData 属性。

```
/****** Flex *****/
imageBitmapData = new ShipGif().bitmapData;
```

## 使用声音

通过两个简单的技巧，能够使得声音 ( 或者音效 ) 能够同时兼容 Flash 和 Flex。首先，通过使用一个 Sound 类而不使用库中的关联名，创建一个 Sound 的实例。这样就能够允许 Flash 和 Flex 同时使用同一个声音素材，尽管内置类型有小小不同。



就像使用这个：

```
public var soundBonus:Sound=new SoundBonus();
```

而不用这个：

```
public var soundBonus:sound_bonus =new SoundBonus();
```

然后，对于 Flex，必须嵌入包含声音的 SWF 文件。

```
[Embed(source = "assets/flackassets.swf", symbol="SoundBonus")]
```

```
private var SoundBonus:Class;
```

如果这些起不到什么作用，不要担心。当我们讨论构成 Flak Cannon 这个游戏的各个类的时候，我们会详细的讲述这些代码的。

## 声音管理器

SoundManager.as 是一个我们即将加到游戏框架中的支持类。这个类的作用不用多说，它是帮助我们管理一切的声音（或者音效）的。为什么我们需要这个类？虽然 ActionScript3.0 有一个简单的 API 去访问声音，但是方法比较复杂，而且不能在一个单独的地方使用他。在 Flash 和 Flex 中嵌入声音的做法是不同的，目前，在 Flex 中，我们会用到之前讲到的 Embed code 方法来做。首先我们要做得是建立一个 SoundManager.as 类，下面是一些需要解释的成员属性：

- sounds : Array，用来存储 SoundManager 要用到的生意。
- soundChannels : Array，用来存储 SoundChannel 对象，我们保留 SoundChannel 对象，这样，当我们接受到事件的时候就可以停止或者改变声音。
- soundTrackChannel : SoundChannel，用来播放指定的声道，在后面我们会讨论一下声道。
- soundMute : Boolean，这个变量用来记录声音是否正在播放。
- tempSoundTransform : SoundTransform，这是个重用的 SoundTransform 对象。
- muteSoundTransform : SoundTransform，用来消去声音的。
- tempSound : Sound，用来存放我们想播放的声音。

我们在构造函数中只初始化 sounds 和 soundChannels

```
package com.efg.framework
{
    import flash.media.*;
    public class SoundManager
    {
        private var sounds:Array;
        private var soundTrackChannel:SoundChannel=new SoundChannel();
```



```
private var soundChannels:Array;
private var soundMute:Boolean = false;
private var tempSoundTransform:SoundTransform = new SoundTransform();
private var muteSoundTransform:SoundTransform = new SoundTransform();
private var tempSound:Sound;
public function SoundManager()
{
    sounds = new Array();
    soundChannels = new Array();
}
}
```

现在我们需要一个能够添加声音到 sounds 数组的方法，我们会通过一个关联数组来完成。用 soundName 作为数组的索引，然后把 sound 对象插入到相应的地方。在游戏中会定义一个 Main 类，soundName 会以这样的形式出现：Main.SOUND\_XXX，并且是 static const 的。我们会在下一章讨论这个。

还有一些可以解决添加声音的方法。既然我们已经选择用 String 来作为数组的索引，能够使这个关联数组有效。作为能够选择的，你都可以使用 int 来作为数组索引，只要和 Main.SOUND\_XXX 的值对应起来。我们选择 String 来作为数组索引是因为这样使得代码更容易理解，但是你必须意识到这并不是解决这个问题的唯一方法。现在很多 Flash 开发人员不屑于使用这种方法，但是我们依然相信它有用。

```
public function addSound(soundName:String, sound:Sound):void {
    sounds[soundName] = sound;
}
```

首先，我们初始化 sounds 数组，之后开始往数组里添加声音。回调在 Flash 中和奖励声音关联好的 SoundBonus 的构造函数 SoundBonus()，之后我们就在数组创建 6 个值和 Main.SOUND\_XXX 中的变量（译者注：这些变量用作索引）关联起来。现在，我们必须创建一个方法，当收到命令的时候能够播放 sounds 数组中的某个声音。我们用 playSound() 方法来完成这个功能。在我们的游戏框架中，当游戏用 CustomEventSound 对象（在下一章定义）抛出事件到 Main 中，并且事件类型参数是 CustomEventSound.PLAY\_SOUND 的时候，就会调用 playSound()。现在，我们来看看 playSound() 中的两个参数。

- soundName : String，要播放得声音的名字，应该是定义在 Main 中的几个 static const 变量中的一个；
- isSoundTrack : Boolean，用来判断这个声音是否当作声道来播放，声道要特殊处理。
- loops : int，用来定义这个声音播放（或者循环）的次数。
- offset : Number，This Number defines the offset (in milliseconds) from which the sound will start playing.



- volume : Number 音量大小 ( 从 0 到 1 )。

这里是 playSound()的定义：

```
public function playSound(soundName:String, isSoundTrack:Boolean=false, loops:int=1,
offset:Number=0, volume:Number=1):void
{
```

这个函数的前两行代码设置了两个临时变量，tempSoundTransform 和 tempSound，tempSound 用来存储 sounds 数组中索引为 soundName(译者注：此函数第一个参数)的声音，tempSoundTransform 就复杂一点，是用来改变音量和 pan Sound 对象的属性。在 Sound 中你没有改变音量和直接 pan，所以你需要这个对象去执行这些功能。

```
tempSoundTransform.volume=volume;
tempSound = sounds[soundName];
```

现在，我们需要根据 playSound()中 isSoundTrack 参数的值来做出决定。如果声音是被定义为一个声道，那就要和其他声音区别对待。我们定义一个声道来循环音乐或者持续播放一段时间。

我们添加这些功能是因为我们发现当播放一个声道，尤其是多个声道不同时间播放的时候，AS3.0 处理起来比较困难。我们需要的功能是这样的：在一个时间只播放一个声道，如果一个声道被设置为播放，那么其他声道就要关闭。这听起来简单，但是当我们使用一个状态机的时候，下面的情况就有可能出现了：

- 你正在标题屏幕播放声道 A。
- 游戏开始的时候你播放声道 B。
- 在结束得屏幕你播放声道 C。
- 当你关闭结束屏幕，回到主题屏幕，声道 A 又播放了。

这个情况得问题是，当主题屏幕第一次播放，没有声音播放，所以我们播放声道 A。然而，当标题屏幕播放第二次（当结束屏幕关闭后），声道 C 已经在播放了，但是我们需要再次播放声道 A。这种情况，非常简单，马上启动两个声道。当结束屏幕关闭或者标题屏幕显示的时候，你可以通过一些逻辑去关闭声道 C，但是这可能失去控制。随着声道数量和游戏状态的增加，播放声道的难度也会提升。不要管理所有的声道，一次只播放一个声道看起来非常有用。

为了减少这些声道功能，我们已经创建了一个单一的 SoundChannel 对象 soundTrackChannel。如果 soundTrackChannel 是非空的（就是说一个声道正在播放），我们就调用 soundTrackChannel 的 stop() 函数，如果是空的，我们就设置 soundTrackChannel 等于我们当前播放的声音，然而，这个代码引出了一个问题：“什么是 SoundChannel”。

AS3.0 中的 SoundChannel 对象是用来区分正在播放的声音，所以你可以为每一个声音单独的设置音量和 pan 属性。我知道这听起来并不强大，但这是 AS2 之后的改进。在 AS2 中你被迫创建一个 Sound 对象绑定到一个 MC 对象来确保声音能够正常运作，代码仍然很多 bug。而且会很有问题，无论我们如何努力尝试，游戏中得每一个声音都是同一个音量，或者更糟糕的，完全没有声音。

有了 SoundChannel 对象，这些限制都消除了，你要付出的代价就是你的代码会比之前的 Flash 版本



稍微复杂一点。好消息是：你每次播放一个声音，都会给你提供一个 SoundChannel。你所要做的事情就是通过一个引用设置它的属性。在我们的代码中，那个引用存储在 soundChannels 数组中，由 soundName 参数标记。我们存储这些 SoundChannel 是有原因的，如果我们不存储的话，我们就不能根据需求来停止某个正在播放的声道。当我们讨论到 stopSound() 的时候会讲到这个的。

```
if (isSoundTrack) {
    if (soundTrackChannel != null) {
        soundTrackChannel.stop();
    }
    soundTrackChannel = tempSound.play(offset, loops);
    soundTrackChannel.soundTransform = tempSoundTransform;
} else {
    soundChannels[soundName] = tempSound.play(offset, loops);
    soundChannels[soundName].soundTransform = tempSoundTransform;
}
}
```

当 Main 收到一个 CustomEventSound 事件且类型是 CustomEventSound.STOP\_SOUND 的时候，就会调用 stopSound()。函数会看我们是否当做一个声道去停止，然后再停止相对应的声音。如果是声道，我们无需理会 soundName，只需要调用 soundTrackChannel 的 stop()。如果不是声道，我们就根据 soundName 去查找 soundChannels 中对应的声音，并停止。

```
public function stopSound(soundName:String, isSoundTrack:Boolean=false):void {
    if (isSoundTrack) {
        soundTrackChannel.stop();
    } else {
        soundChannels[soundName].stop();
    }
}
```

函数 muteSound() 是用来关闭所有游戏中播放的声音，就像是一个 sound channel，我们需要 SoundTransform 对象去设置音量来关闭所有声音（译者注：普通声音和声道）。首先查看

soundMute 的值，如果为真，如果是我们已经关闭的声音，我们就重新打开，如果为假，则关闭一切声音。关闭声音的方法是，设置 muteSoundTransform.volume 的值为 0。打开的话，就设置为 1。然后，我们设置 flash.media.SoundMixer.soundTransform 的静态属性为 muteSoundTransform。这正好运用了 Flash 的长处去播放声音，可以全程控制音量。我们还设置 soundMute 为 true 或 false，具体取决于我们切入的哪个部分的 if 语句，这样，下次调用这个函数的时候我们就知道要做什么了。

```
public function muteSound():void {
    if (soundMute) {
        soundMute = false;
        muteSoundTransform.volume = 1;
    }
}
```



```
flash.media.SoundMixer.soundTransform=muteSoundTransform;
}else{
    muteSoundTransform.volume=0;
    flash.media.SoundMixer.soundTransform=muteSoundTransform;
    soundMute=true;
}
}
```

我知道这比简单地播放声音要复杂得多，但是在这本书的后面我们会重用（和扩展）这个类。然而，我们还没有完成。我们还需要创建自定义的事件类来播放声音和停止声音。好消息是，这个事件类非常像我们在第二章创建的事件类。不同之处是我们这里有两个不同得事件类型（PLAY\_SOUND 和 STOP\_SOUND），和在事件抛出的时候我们需要设置一个参数。

- name : String，是 Main 类的静态常量，用来表示我们想要播放的声音，在下一章我们会定义我们的常量。
- loops : int，定义声音循环的次数，对于一个声道来说，这个值可能会很大。
- offset : Number，offset 的毫秒数。
- volume : Number，表示音量大小，在 0 到 1 之间。
- isSoundTrack : Boolean，如果这个值为真，则 SoundManager 会把它当作声道来处理，并会用专门的 SoundTrackchannel 来播放。

下面我们开始定义 CustomEventSound 类：

```
package com.efg.framework
{
    import flash.events.*;
    public class CustomEventSound extends Event
    {
        public static const PLAY_SOUND:String = "playsound";
        public static const STOP_SOUND:String = "stopsound";
        public var name:String;
        public var loops:int;
        public var offset:Number;
        public var volume:Number;
        public var isSoundTrack:Boolean;
        public function CustomEventSound(type:String,name:String,
            isSoundTrack:Boolean=false, loops:int=0,offset:Number=0,
            volume:Number=1, bubbles:Boolean=false,cancelable:Boolean=false) {
            super(type, bubbles, cancelable);
            this.name = name;
        }
    }
}
```



```
        this.loops = loops;
        this.offset = offset;
        this.volume = volume;
        this.isSoundTrack = isSoundTrack;
    }
    public override function clone():Event {
        return new CustomEventSound(type, name,isSoundTrack,loops,offset,
        volume,bubbles,cancelable)
    }
    public override function toString():String {
        return formatToString(type, "type", "bubbles", "cancelable",
        "eventPhase",name,isSoundTrack,loops,offset,volume);
    }
}
```

现在,我们已经完成了一个全功能的 SoundManager,可以添加到游戏框架中被我们的游戏使用。Flak Cannon 可能不需要用到我们创建的所有功能,但是我们已经完成了这个类,这样能够应付到更为复杂的声音应用。

## 设置难度

优化一个游戏的难度是一件很消耗时间的东西,而且有很多方法可以解决这个问题。在这个游戏,我们会采用 Rob Fulop 设计的一个即兴游戏的一个方法。Fulop 是经典的 Atari VCS 的其中一个开发人员。他编程了 Missile Command 的 Atari VCS 版本,但是被人知道是因为 Demon Attack,这个游戏被认为是最好的 Atari VCS 游戏之一。Fulop 的建议是制作尽可能多的旋钮,这样就可以调节到游戏的不同部分以满足你的需求。由于我们不会真的制作旋钮,我们可以认为是各种设置。这些设置可以有很多形式,但是对于这个游戏,他们会简化为数字变量,当一个新的难度创建的时候就会更新,我们就会插入到我们的代码中。

## 难度设定

我们会用一组变量来实现这些设定,测试用来更新游戏当前难度的设置值。大部分的难度设定使用下面的简写: if:then format:

value = test:? 为真的表达式: 为假的表达式

这种格式可以很容易地创建一个限额定在那里设置,如果达成,默认为设定值。这样做的话,游戏的最高分就会出现在一个比较大的难度。现在我们已经准备好开始设置了。

- numEnemies:这是用来设置每个难度出现的敌方飞机的数量,最多是 100。一开始是 15,然后是 30,如此类推。



```
numEnemies =(numEnemies > 100) ? 100 :level * 10 + (level*5);
```

- enemyWaveDelay : 这是用来设置下一波敌人出现的等待时间。这个时间越短，游戏难度越大。最小是 20 帧，一开始是 60，然后是 58，56，如此类推。

```
enemyWaveDelay =(enemyWaveDelay < 20)? 20:60 - (level-1)*2;
```

- enemyWaveMax : 这是用来设置一波攻击中同时出现的敌方飞机的最大数量。最大值是 8，一开始是 2，然后是 3,4，如此类推。在下面的代码，我们乘以 1，为什么呢？其实我们没有必要那样做，乘以 1 只是用来占个位置而已，如果你想很明显的增加敌方飞机，你可以通过乘以一个更大的数字来达到目的。

```
enemyWaveMax =(enemyWaveMax > 8) ? 8:1 * level+1;
```

- enemyWaveMultipleChance : 这个值是用来设置一波攻击中有多架飞机出现的几率。一开始是 10，每升一级就提高 10。

```
enemyWaveMultipleChance =(enemyWaveMultipleChance > 100)? 100:10*level;
```

- enemySpeed : 这是用来设置敌方飞机的飞行速度的（就是每一帧移动的像素）。最大是 8 一开始是 2，然后 2.5，3，3.5 如此类推。

```
enemySpeed =(enemySpeed > 8) ? 8:2 + (.5*(level-1));
```

- enemySideChance : 这是用来设置敌方飞机从屏幕侧面飞过来的几率。最大是 70，一开始是 0，每一级提升 10。

```
enemySideChance =(enemySideChance > 70)? 70:10*(level-1);
```

- enemySideFloor : 这是敌方飞机从侧面最靠近玩家守卫的舰队的进入的地方。

```
enemySideFloor =(enemySideFloor > 300)? 300:100 + 25*(level-1);
```

- bonusPlaneDelay : 这是设置“奖励飞机”多长时间飞出来一次的。

```
bonusPlaneDelay =(bonusPlaneDelay > 450)? 450:350 + 10*(level-1);
```

- bonusPlaneSpeed : 这是设置“奖励飞机”飞行速度的。

```
bonusPlaneSpeed =(bonusPlaneSpeed > 12) ? 12 : 4 + (1*level);
```

- startShots : 这是设置玩家一开始拥有的弹药量，是 30。
- extraScore : 这是用来设置一个分数的，当玩家的分数到达这个值的话就可以额外获得一个军舰，是 10000。
- baseEnemyScore : 这是用来设置机会一架敌方飞机可获得的分数，是 100。
- enemyHitBonus : 这是用来设置一个弹药击毁多架敌方飞机所获得的分数，是 500。
- baseBonusPlaneScore : 这是用来设置击毁一架“奖励飞机”所获得的分数，是 500。



- `maxBonusPlanes` : 置在同一时间内屏幕上的“奖励飞机”最多的数量, 是 1。
- `maxVisibleShips` : 这是用来设置在任何时候屏幕上所能够显示的军舰数目, 是 3。
- `bonusPlaneMaxY` : 这是用来设置“奖励飞机”在屏幕中最高的位置, 是 350。
- `shipYPadding` : 这是用来设置军舰底部和屏幕底部之间的像素距离, 是 5。

我们会在下一章的 `newLevel()` 函数中设置这些变量。现在, 你只需要理解我们会用这些设置去设置游戏里的难度。它们也有利于你通过代码来开始体验和运用它来润色你的游戏。你创

*建越多的设置, 你就越容易在你的游戏中控制每个等级难度的平衡。不过, 同时也有可能代码中制造了更复杂的逻辑和 bug。作为一个游戏开发者, 你就需要在两者之间寻求一个合适的度了。*

## 创建场景

我们现在需要创建静态场景。虽然这些场景有可能移动, 但是他们只有一帧动画。在我们的游戏中, `Shot`, `Enemy`, `BonusPlane`, 和 `Ship` 对象都是静态 `sprite`, 我们用一种比较简单的方式定义他们。

## 定义一个在两点间移动的 `sprite`

这里有个问题, 如果你有两个点, 那你能做的最基本的事情是什么? 我想, 还是有很多事情可以做的, 不过最重要的应该是: 画一条直线。这看起来好像不是很有用, 不过我们可以通过一条直线来决定两点之间的距离。如果我们知道两点之间的距离和物体沿着这条直线移动的方向和速度, 我们就可以算出在 Flash 中一帧要移动多少。好消息是我们可以决定这一切 *on the fly* 和使 `sprite` 从一点移动到另一点。现在先看看 `Shot` 类的简单设计。

### `Shot`

我们看到的第一个 `sprite` 是 `Shot`。它是用来呈现玩家点击鼠标发射炮弹效果的。`Shot` 总是从屏幕的底部开始发射, 一直飞到十字准星的中心。这意味着我们总有两个点需要处理。这非常重要, 因为这两个点会使得我们的 `sprite` 移动起来很方便。首先, 我们需要导入必要的类来处理 `Bitmap` 图像和 `BitmapData`, 还要加上基础类 `Sprite`。

```
package com.efg.games.flakcannon
{
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.geom.Point;
```



```
public class Shot extends Sprite
{
}
```

然后，我们定义类中需要用到的变量。首先，我们需要一个变量去保存 ShotGif 图像，还有相关的 BitmapData。或者，你可以在 Flash IDE 中把 Shot 当做一个 MovieClip 来创建，然后拖一个 bitmap 图像到它上面，我们要重点做好这一步，这样我们就可以把代码移植到其他开发工具中。我们会用一个 image 变量来保存 ShotGif 实例，一个 imageBitmapData 变量来保存 ShotGif 的 BitmapData。

```
public var imageBitmapData:BitmapData;
public var image:Bitmap;
```

下一步，我们要创建两个变量来记录两个点，是用来计算 Shot 的移动向量的（包括方向和大小）。最简洁的方法就是用 Point 对象了。我们给这两个点起个名字吧，一个是 startLocation，一个是 endLocation。

```
private var startLocation:Point;
private var endLocation:Point;
```

接下来，我们需要一对变量来记录在一帧里，Shot 移动到一个坐标是 (x, y) 的飞机需要的量。我们会用 xunits 和 yunits 来保存。

```
private var xunits:Number;
private var yunits:Number;
```

在实际移动 sprite 之前，我们会先计算 sprite 下一步会移动到哪里。我们用一个叫 nextLocation 的 Point 对象来保存下一步的坐标。

```
private var nextLocation:Point;
```

现在我们需要设置 Shot 的速度（一帧 15 个像素），还要创建一个变量来记录 Shot 到达目的地前移动的距离。同时，我们也需要一个判断 Shot 是否到达目的地的变量，我们用 finished，一个 Boolean 变量来判断。

```
private var speed:int = 15;
private var moves:int = 0;
public var finished:Boolean;
```

最后，如果我们是在 Flex 中创建游戏的话，我们需要导入 ShotGif 图片。你需要反注释下面的这些代码：

```
/*
/**Flex Framework Only
[Embed(source = "assets/flakassets.swf", symbol="ShotGif")]
private var ShotGif:Class;
*/
```

Shot 的构造函数非常简单，包含两个点，(startX, startY)和(endX, endY)，并用我们的成员变量去保



存 startLocation 和 endLocation 。我们都会初始化 nextLocation , 方便我们后面使用。

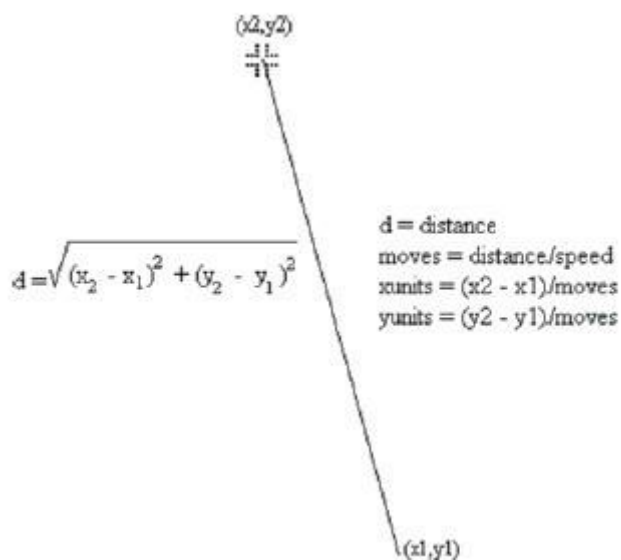
之后我们调用 init ( ) , 有趣的事情发生了。

```
public function Shot (startX:Number, startY:Number, endX:Number, endY:Number)
{
    startLocation = new Point(startX,startY);
    endLocation = new Point(endX,endY);
    nextLocation = new Point(0,0);
    init();
}
```

我们只是定义了 Shot 类中很基础的一部分。现在, 我们填充它需要的功能。好消息是, 虽然 Flak Cannon 中有很多会移动的对象, 但是这些对象大多使用相似的代码, 所以我们只需要编写一次就够了。我们会在 init()中计算 Shot 的 xunit 和 yunit , 我们第一件要做的事情就是计算 startLocation 和 endLocation 之间的距离。

```
public function init():void
{
    x = startLocation.x;
    y = startLocation.y;
    var xd:Number = endLocation.x - x;
    var yd:Number = endLocation.y - y;
    var distance:Number = Math.sqrt(xd*xd + yd*yd);
```

距离变量有一个标准的几何公式, 就像是图【4-14】所示。我们把公式转换为 AS3 代码, 不过还是同样的计算。



图【4-14】用公式来计算 xunits 和 yunits 的值

现在我们能够计算起点到终点连线的长度，如果我们知道 Shot 速度的话，我们就可以计算出从起点到终点需要多少帧了。很幸运，我们已经知道 Shot 的速度了。

```
moves = Math.floor(Math.abs(distance/speed));
```

最终，通过计算 x 方向的移动(endLocation.x - x) 和 y 方向的移动(endLocation.y - y)就可以计算出每一帧要移动的像素的准确值。

```
xunits = (endLocation.x - x)/moves;
yunits = (endLocation.y - y)/moves;
```

进行一次这样的计算的好处是，这个 Shot 在游戏中剩下的时间里，通过一些简单的运算就可以进行移动了。所有要用到的变量都已经计算好了。当然也有其他很多的方法可以计算一个向量和移动一个 sprite，不过这是目前为止我们知道的在一条线上移动一个对象到一个已知的点的最好方法。在我们讲到类 Enemy 的时候，我们会讨论一个比较普遍使用的移动对象算法。

现在，我们必须在库中获取 ShotGif 图片，并把图片关联到一个 Bitmap。(记住，在 Flex 中，你需要反注释掉 Flex 代码和注释掉 Flash 代码)

```
//***** Flex *****
//imageBitmapData = new ShotGif().bitmapData;
//***** Flash *****
imageBitmapData = new ShotGif(0,0);
image = new Bitmap(imageBitmapData);
```

最后，我们把 image 添加到 Shot 的一个实例中，并把成员变量 finished 设为 false。

```
addChild(image);
finished = false;
```



现在是时候编写类 Shot 的 update(), render()和 dispose()方法了。update()函数被 Game 类 (FlakCannon.as)调用用来设置 nextLocation。我们在 update()中并不做任何实际的操作。理由很简单,如果我们做任何形式的预先碰撞检测(例如,墙壁碰撞测试),我们会希望在事前确保获得所有的 nextLocation 值。即使我们在游戏中不使用预先碰撞检测,我们都会用相同的约定。

在 update()中第一件做的事就是查看 moves 是否为 0,如果为 0,我们就把 finished 设为 true,下一次 Game 检测 Shot 已经完成了,炮弹就会爆炸。这个处理我们后面会讨论。我们就更新

nextLocation 和减少 moves.

```
public function update():void {  
    if (moves > 0) {  
        nextLocation.x = x + xunits;  
        nextLocation.y = y + yunits;  
        moves--;  
    } else {  
        finished = true;  
    }  
}
```

我们告诉你 update()只是用来设置新的 moves 的值。render()方法是真的要作处理的,虽然在这里很简单。我们用 x,y 值去设置 nextLocation 的 x,y 值。像 update()一样,render()也是被 Game 调用的。

```
public function render():void {  
    x = nextLocation.x;  
    y = nextLocation.y;  
}
```

最后,我们来编写 dispose()函数,当一个 Shot 到达目的地并爆炸之后,我们就需要移除这个 Shot 了。Game 类会负责把 Shot 从屏幕中删除,并从 Shot 数组中剔除。dispose()函数则负责在内存中移除 Bitmap 和 BitmapData。虽然这个操作不是必须的,但是有助于 AS3 的垃圾回收。如果不这样做的话,由于可用的内存逐渐减少,游戏会变得越来越慢。

```
public function dispose():void {  
    removeChild(image);  
    imageBitmapData.dispose();  
    image = null;  
}  
}
```

上面说的就是 Shot 类的全部了。不过,我们还需要继续说说 Enemy, Ship,和 BonusPlane。很幸运, BonusPlane 和 Shot 几乎是一样的,所以我们接下来就先看看 BonusPlane, Enemy 和 Ship



也和 Shot 有相同之处。你可能会想，既然这些类有那么相同之处，我们为什么不写一个类，然后再继承呢？诚然，那样会好点，我们没有使用继承是为了你能够更好的理解。你可以很容易

从这些类中抽象出相同的共性，并且创建一个基类来让去实例化这些对象。这个就留着你自己做了，这样你就可以按着你的意思去创建一个更复杂的对象了。

## BonusPlane

我们会利用 Shot 的代码来创建 BonusPlane。其中需要改动的代码非常少。BonusPlane 总是在屏幕上从左往右飞过。最大的不同就是 bonusValue 这个成员变量。bonusValue 是用来记录玩家通过击中“奖励飞机”获得的弹药。

```
package com.efg.games.flakcannon
{
    import flash.display.Shape;
    import flash.display.Sprite
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.geom.Point;
    public class BonusPlane extends Sprite
    {
        public var imageBitmapData:BitmapData;
        public var image:Bitmap;
        private var startLocation:Point;
        private var endLocation:Point;
        private var nextLocation:Point;
        private var xunits:Number;
        private var yunits:Number;
        private var speed:Number = 5;
        private var moves:int = 0;
        public var bonusValue:int = 0;
        public var finished:Boolean;
        /*
        [Embed(source = "assets/flakassets.swf", symbol="PlaneBonusGif")]
        private var PlaneBonusGif:Class;
        */
        public function BonusPlane(startX:Number, startY:Number, endX:Number, endY:Number, □
        speed:Number, bonusValue:int) {
            startLocation = new Point(startX,startY);
            endLocation = new Point(endX,endY);
            nextLocation = new Point(0,0);
        }
    }
}
```



```
this.speed=speed;
this.bonusValue=bonusValue;
init();
}
```

当 init 被调用的时候，唯一不同的地方就是用来创建图片的 Bitmapdata。这次用到的函数是 PlaneBonusGif().

```
public function init():void
{
    x = startLocation.x;
    y = startLocation.y;
    var xd:Number = endLocation.x - x;
    var yd:Number = endLocation.y - y;
    var Distance:Number = Math.sqrt(xd*xd + yd*yd)
    moves = Math.floor(Math.abs(Distance/speed));
    xunits = (endLocation.x - x)/moves;
    yunits = (endLocation.y - y)/moves;
    //***** Flex *****
    //imageBitmapData = new PlaneBonusGif().bitmapData;
    //***** Flash *****
    imageBitmapData = new PlaneBonusGif(0,0);
    image = new Bitmap(imageBitmapData);
    addChild(image);
    finished = false;
}
```

BonusPlane 剩下的功能函数和 Shot 是一样的：

```
public function update():void {
    if (moves > 0) {
        nextLocation.x = x + xunits;
        nextLocation.y = y + yunits;
        moves--;
    } else {
        finished = true;
    }
}
```

```
public function render():void {
    x = nextLocation.x;
    y = nextLocation.y;
```



```
}  
public function dispose():void {  
    removeChild(image);  
    imageBitmapData.dispose();  
    image = null;  
}  
}
```

## Ship

现在我们来开始讨论 Ship 对象。

Ship 是 Flak Cannon 里面最简单的 Sprite 类，因为它不会移动。他们真的是静态 sprites！它们只是静静地呆在那里，等待被攻击（呃，被守护）。那就是说，和 Shot 最大的不同之处就是使用 ShipGif() 来获得我们要的 BitmapData。

```
package com.efg.games.flakcannon  
{  
    import flash.display.Shape;  
    import flash.display.Sprite;  
    import flash.events.MouseEvent;  
    import flash.display.Bitmap;  
    import flash.display.BitmapData;  
    public class Ship extends Sprite  
    {  
        public var imageBitmapData:BitmapData;  
        public var image:Bitmap;  
        /*  
        /**Flex Framework Only  
        [Embed(source = "assets/flakassets.swf", symbol="ShipGif")]  
        private var ShipGif:Class;  
        */  
        public function Ship() {  
            init();  
        }  
        public function init():void {  
            /******* Flex *****  
            //imageBitmapData = new ShipGif().bitmapData;  
            /******* Flash *****  
            imageBitmapData = new ShipGif(0, 0);  
            image = new Bitmap(imageBitmapData);  
        }  
    }  
}
```



```
        addChild(image);
    }
    public function dispose():void {
        removeChild(image);
        imageBitmapData.dispose();
        image = null;
    }
}
}
```

## 创建在一个连续向量上移动的对象：Enemy

不同于 BonusPlane 和 Ship，Enemy 和 Shot 还是有很不同的地方。有些 Enemy 是来自一个模拟了 3 种不同种类的 Enemy 的类中。我们这样做是为了支持 Enemy 飞机能够有 3 个不同的方向：DIR\_DOWN, DIR\_RIGHT, or DIR\_LEFT。还有一个更为实质性的改变。我们本来应该像创建 Shot 和 BonusPlane 的移动那样去创建 Enemy 的移动方式。我们没有这样做，我们打算介绍一个关于移动的新算法，能够支持对象能够向任何方向移动。

一开始，我们先来快速浏览一下类的成员变量。注意，我们添加了 4 个新的变量。dir 是用来表示 Enemy 飞机飞行的方向的，DIR\_DOWN, DIR\_RIGHT 和 DIR\_LEFT 中一个来表示。因为这三个变量是静态变量，所以在 Enemy 类外也可以访问。

```
package com.efg.games.flakcannon
{
    import flash.display.Shape;
    import flash.display.Sprite
    import flash.events.MouseEvent;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.geom.Point;
    public class Enemy extends Sprite
    {
        public var imageBitmapData:BitmapData;
        public var image:Bitmap;
        private var startLocation:Point;
        private var endLocation:Point;
        public var nextLocation:Point;
    }
}
```

注意，nextLocation 在类中式一个 public 变量。这样，在碰撞检测的时候就可以访问到它。由于在游戏中我们不会用到预先碰撞检测，我们也不打算用到这个特性，不过这是一个我们值得注

意的地方。你需要记住，使变量能够从外部访问，既可以直接把变量声明为 public，也可以利用 get/set



方法。

另一个添加的变量是 angle。我们正在创建一个从一点出发，能够向各个方向移动的对象，直到我们要求它停下。而不是从一点移动到另一点，然后停下来。为了方便起见，我们会保存当前

移动对象的角度。这个值会帮助我们在 reader()中计算 nextLocation 的值。

```
private var speed:int = 5;
public var finished:Boolean;
public var dir:Number;
public var angle:Number;
```

```
public static const DIR_DOWN:int = 1;
public static const DIR_RIGHT:int = 2;
public static const DIR_LEFT:int = 3;
```

如果是使用 Flex 的话，要注意了，我们需要嵌入 3 张图片。这 3 张图片是用来展现 Enemy 飞机的几个飞行方向。

```
/*
/**Flex Framework Only
[Embed(source = "assets/flakassets.swf", symbol="PlaneGif")]
private var PlaneGif:Class;

[Embed(source = "assets/flakassets.swf", symbol="PlaneLeftGif")]
private var PlaneLeftGif:Class;

[Embed(source = "assets/flakassets.swf", symbol="PlaneRightGif")]
private var PlaneRightGif:Class;
*/
```

在 Enemy 的构造函数中也有不同，我们从 FlakCannon 传递一个 speed 参数过来，而不是在构造函数中设置 speed。这就允许 Game 在游戏难度升级的时候使用 enemySpeed 难度 setting 去设置

Enemy 飞机的飞行速度。我们也给 dir 传一个参数。这就允许 Game 在游戏难度提升的时候使用 enemySideChance 难度设置。

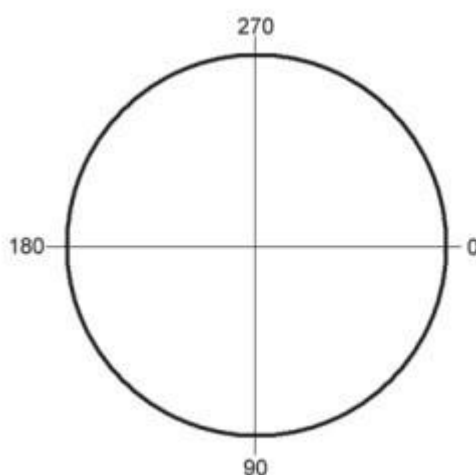
```
public function Enemy(startX:Number, startY:Number, endY:Number, speed:Number,dir:int) {
    startLocation = new Point(startX,startY);
    endLocation = new Point(0,endY);
    nextLocation = new Point(0,0);
    this.dir = dir;
    this.speed=speed;
```



```
init();  
}
```

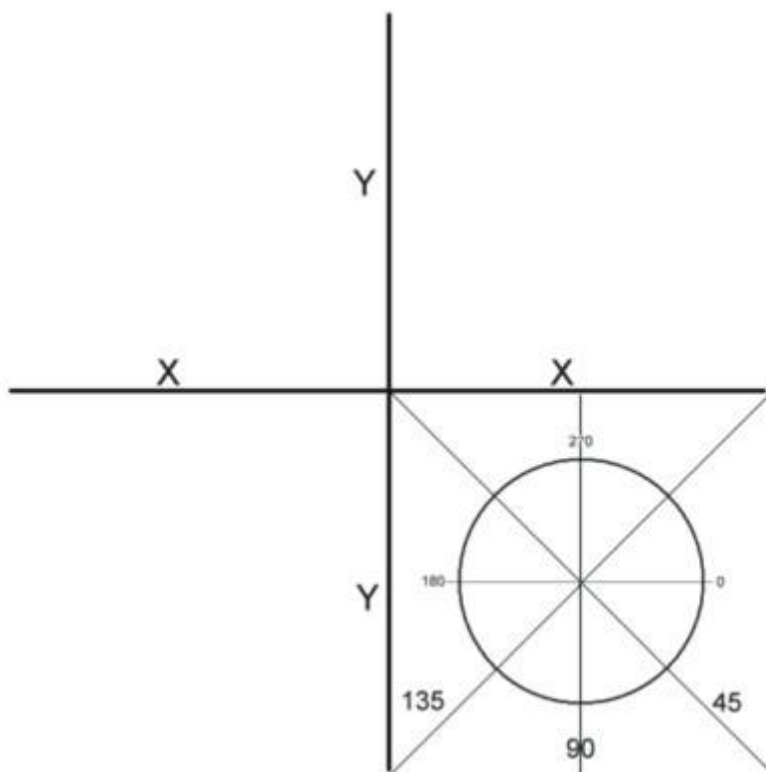
最后,我们需要对 dir 成员变量做一些处理。在 init() 中,我们使用 switch,根据 dir 的值去选择 Enemy 飞机的图片,这里,Flash 和 Flex 的代码会有不同。然而,比起我们之前创建的 Shot 和 BonusPlane,这个函数会有更重要的不同之处。不再使用已经计算好的 xunits 和 yunits,这个函数通过 speed 和 angle 这两个变量来解决向量的问题。这样处理向量的问题会更加灵活,因为你可以通过改变角度来计算向量移动。

一开始,我们先来讨论 Flash 是怎样处理角度的。Flash 使用一个二维坐标来放置对象。我们已经用这种方式给很多 Sprite 和 MovieClip 定好位置了。当我们使用角度,详细了解坐标系统是怎样处理就显得非常重要了。Angles in Flash are calculated with the origin of 0 degrees pointing directly to the right of the screen.



图【4-15】Flash 怎么计算角度。

在图【4-16】我们可以看到 2D 飞机在游戏中是怎样放置的。x 和 y 的值越大,飞机的位置就越靠近右下。这就意味着,一个 45 度角,会像图中那样从中间“剪开”。同样,一个 0 度角会比这条线少 45 度。



图【4-16】Flash 中的笛卡尔坐标

在屏幕中，旋转 90 度就指向屏幕正下方了。由于 Enemy 飞机会向下飞，所以我们需要特别注意这个值。同样，我们也要注意 45 度和 135 度这两个角度，因为 Enemy 飞机会从这两个方向飞进屏幕。

```
public function init():void
{
    x = startLocation.x;
    y = startLocation.y;
    switch(dir) {
        case DIR_DOWN:
            //***** Flex *****
            //imageBitmapData = new PlaneGif().bitmapData;
            //***** Flash *****
            imageBitmapData = new PlaneGif(0,0);
            angle = 90;
            break;
        case DIR_RIGHT:
            //***** Flex *****
            //imageBitmapData = new PlaneRightGif().bitmapData;
            //***** Flash *****
```



```
        imageBitmapData = new PlaneRightGif(0,0);
        angle = 45;
        break;
    case DIR_LEFT:
        //***** Flex *****
        //imageBitmapData = new PlaneLeftGif().bitmapData;
        //***** Flash *****
        imageBitmapData = new PlaneLeftGif(0,0);
        angle=135;
        break;
    }
    image = new Bitmap(imageBitmapData);
    addChild(image);
    finished = false;
}
```

现在我们已经有了角度和速度了，可以计算 Enemy 飞机的下一个坐标了。调回那个需要速度和角度，然后把需要的值填上去。我们首先要做的事情就是把角度转换为弧度。因为弧度是计算三角函数 sin,cos 的标准单位。它只是在我们运用 sin 和 cos 来计算 Enemy 飞机的 nextLocation 才用到。你只需要知道角度怎样转换为弧度。

公式：弧度 = 角度 \* Math.PI / 180。

为了计算 nextLocation，我们会利用 sin()去计算 y 的改变，cos()去计算 x 的改变。然后再加上当前的 x,y 值。这样就得到 nextLocation 的值了。

```
public function update():void {
    if (y < endLocation.y) {
        var radians:Number = angle * Math.PI / 180;
        nextLocation.x = x + Math.cos(radians) * speed;
        nextLocation.y = y + Math.sin(radians) * speed;
    } else {
        finished = true;
    }
}
```

虽然上面关于计算的描述有点复杂，不过对于各个平台和编程语言来说，这几乎是即插即用的。一旦你知道如何使用弧度，sin(),cos()去计算 nextLocation，剩下的东西就简单了。这个方法对比点到点，优势在于我们可以很容易改变 Enemy 飞机的角度，让它从不同的方向飞过来。Enemy 中剩下的方法和 Shot 中的没有本质的区别。

```
public function render():void {
    x = nextLocation.x;
```



```
y = nextLocation.y;
}
public function dispose():void {
    removeChild(image);
    imageBitmapData.dispose();
    image = null;
}
}
}
```

## 创建动画

在 Flak Cannon 中，有两个游戏元素需要多帧动画：Flak 炸弹和爆炸。它们几乎是一样的，但是在移动上有一些很主要的区别。Flash 可以让程序员通过使用内建的时间轴和 MovieClip 来创建动画。在第一章中，我们就是用这个方法创建了一个动画。不过，由于这章是我们用 bitmap 制作游戏的开始，我们打算走捷径。用 MovieClip 去创建动画很简单，不过当你使用它们的时候要付出代价。bitmap 更加快速和高效，在书中，我们会一直用 bitmap 来制作游戏。对一个动画 Sprite 来说，我们需要用到很多图片才能达到我们想要的效果。很幸运，我们已经导入了 7 张用来制作爆炸动画的图片了。现在我们就要寻求一个方法，可以逐张显示，并控制一张图片显示时间的长度。

### Flak

回调我们加入到库中的 flak 爆炸动画（看图【4-3】）。那些就是我们要用来显示爆炸效果的帧。一开始，Flak 的导入和 Shot 的一样。

```
package com.efg.games.flakcannon
{
    // Import necessary classes from the flash libraries
    import flash.display.Shape;
    import flash.display.Sprite
    import flash.events.MouseEvent;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    public class Flak extends Sprite
    {
```

不过，在我们开始定义成员变量的时候，就开始不同了。首先，不再用一张图片了，而是要用一个 images 的数组。用来存放显示爆炸的图片。我们也需要一个 image 变量去保存当前的帧动画，同样，我们也需要 currentIndex(当前图片索引)，用来保存 images 数组中正在显示的图片索引。还要一个 finished 布尔变量，这样，FlakCannon 就可以告诉我们，当爆炸动画结束的时候，我们就可以把这个动画去掉了。frameCounter 用来定义从我们改变图片以来播放的帧数。我们用 frameCounter 和 frameDelay 以来来



定义动画的长度。最后，有一个变量 `hits`，用来记录这个爆炸摧毁的飞机数目。通过判断 `hits`，我们可以给玩家添加奖励。有了这些变量，我们要 `embed` 这些爆炸动画的帧到 `Flex` 就足够了。

```
public var images:Array;
public var image:Bitmap;
public var currentIndex:int = -1;
public var finished:Boolean;
private var frameCounter:int = 0;
private var frameDelay:int = 2;
public var hits:int;

/*
/**Flex Framework Only
[Embed(source = "assets/flakassets.swf", symbol="Exp1Gif")]
private var Exp1Gif:Class;

[Embed(source = "assets/flakassets.swf", symbol="Exp2Gif")]
private var Exp2Gif:Class;

[Embed(source = "assets/flakassets.swf", symbol="Exp3Gif")]
private var Exp3Gif:Class;

[Embed(source = "assets/flakassets.swf", symbol="Exp4Gif")]
private var Exp4Gif:Class;

[Embed(source = "assets/flakassets.swf", symbol="Exp5Gif")]
private var Exp5Gif:Class;

[Embed(source = "assets/flakassets.swf", symbol="Exp6Gif")]
private var Exp6Gif:Class;

[Embed(source = "assets/flakassets.swf", symbol="Exp7Gif")]
private var Exp7Gif:Class;
*/
```

由于我们不需要移动 `Flak` 炸弹，构造函数只是简单的设置 `hits` 为 0 和调用 `init()`。`hits` 是一个简单的分数计算器。由于一个 `Flak` 爆炸不会在摧毁一个 `Enemy` 飞机就消失，所以它可以摧毁很多 `Enemy` 飞机。`hits` 是用来记录一个 `Flak` 炸弹炸毁多少架 `Enemy` 飞机的，一个 `Flak` 炸毁多架 `Enemy` 飞机，就可以获得奖励。

```
public function Flak() {
    hits = 0;
```



```
init();  
}
```

Flack 更多有趣的代码在 init()中。我们打算把 image 创建为一个 Bitmap，并把它加到 Sprite 上。然后，会把所有爆炸动画中用到的图片加到 images 数组中。再次注意，Flash 和 Flex 代码的不同。

然后我们调用 setNextImage()来给当前的 image 获取 BitmapData。把 frameCounter 初始化为 0，finished 为 false。

```
public function init():void  
{  
    image = new Bitmap();  
    addChild(image);  
    /****** Flex *****/  
    /*images = [new Exp1Gif().bitmapData,  
    new Exp2Gif().bitmapData,  
    new Exp3Gif().bitmapData,  
    new Exp4Gif().bitmapData,  
    new Exp5Gif().bitmapData,  
    new Exp6Gif().bitmapData,  
    new Exp7Gif().bitmapData  
    ];  
    */  
  
    /****** Flash *****/  
    images = [new Exp1Gif(32,32),  
    new Exp2Gif(0,0),  
    new Exp3Gif(0,0),  
    new Exp4Gif(0,0),  
    new Exp5Gif(0,0),  
    new Exp6Gif(0,0),  
    new Exp7Gif(0,0)  
    ];  
    setNextImage();  
    frameCounter=0;  
    finished=false;  
}
```

setNextImage()使 currentIndex 递增 1，并判断动画是否结束了。如果是的话，就把 finished 设为 true，如果不是的话，我们就把 image.bitmapData 的值设为 images 数组中索引为 currentIndex (译者注：这时候 currentIndex 已经+1了)的图片。

```
public function setNextImage():void {
```



```
currentImageIndex++;  
if (currentImageIndex > images.length-1) {  
    finished = true;  
} else {  
    image.bitmapData = images[currentImageIndex];  
}  
}
```

update()函数不需要移动任何东西。当它被 FlakCannon 调用，他只需要简单的更新一下 frameCounter。在 render()中有一些有趣的东西。

```
public function update():void {  
    frameCounter++;  
}
```

render()是用来控制动画流程的。首先，它会检查 frameCounter 是否已经到达 frameDelay。如果是的，就调用 setNextImage() 并把 frameCounter 置 0 **we can start counting frames for the next image in the images array.**

```
public function render():void {  
    if (frameCounter >= frameDelay && !finished) {  
        setNextImage();  
        frameCounter=0;  
    }  
}
```

和 Shot 一样，dispose()用来处理那些不用的对象，这样有助于垃圾回收。这里不同的地方是，我们需要处理的是一个数组，而不是一个对象。我们在这里使用一个 for...each 循环(而不是 for 循环)，for...each 是 AS3 中的新内容，不过在其他的编程语言已经出现好久了。for...each 的好处是代码看起来会简洁点，要处理的是数组中的对象(tempImage)。下面要注意了，

tempImage.dispose()中的 dispose()函数是 AS3 中 BitmapData 的内建函数，不是我们自己写的。我们把函数改成同样的名字，是因为函数是有同样的功能的。

```
public function dispose():void {  
    removeChild(image);  
    for each ( var tempImage:BitmapData in images ) {  
        tempImage.dispose();  
    }  
    images = null;  
}
```



## 爆炸

因为爆炸和 Flak 炸弹的效果几乎一样。我们只讨论他们的不同之处。不过，除了图片的设定和数量不同之外，就是 Explosion 没有 hits 成员变量。Explosion 只是一个简单的动画播放，没有其他内容了。Explosion 有五帧动画。图【4-7】。

这里是 Explosion 的代码：

```
package com.efg.games.flakcannon
{
    // Import necessary classes from the flash libraries
    import flash.display.Shape;
    import flash.display.Sprite
    import flash.events.MouseEvent;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    public class Explosion extends Sprite
    {
        public var images:Array;
        public var image:Bitmap;
        public var currentIndex:int = -1;
        public var finished:Boolean;
        private var frameCounter:int = 0;
        private var frameDelay:int = 1;
        /*
        /**Flex Framework Only
        [Embed(source = "assets/flakassets.swf", symbol="Ex21Gif")]
        private var Ex21Gif:Class;
        [Embed(source = "assets/flakassets.swf", symbol="Ex22Gif")]
        private var Ex22Gif:Class;
        [Embed(source = "assets/flakassets.swf", symbol="Ex23Gif")]
        private var Ex23Gif:Class;
        [Embed(source = "assets/flakassets.swf", symbol="Ex24Gif")]
        private var Ex24Gif:Class;
        [Embed(source = "assets/flakassets.swf", symbol="Ex25Gif")]
        private var Ex25Gif:Class;
        */
        public function Explosion() {
            init();
        }
        public function init():void {
```



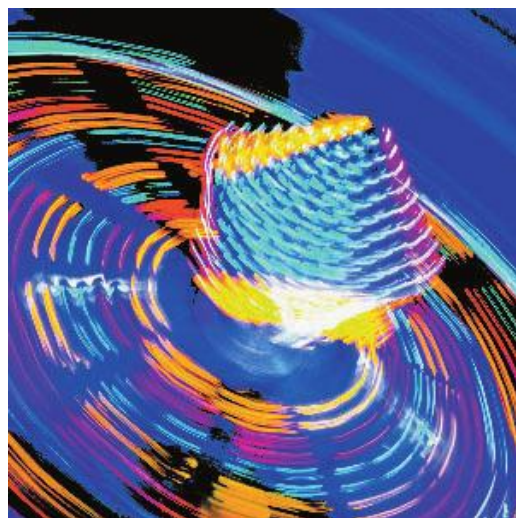
```
image = new Bitmap();
this.addChild(image);
//***** Flex *****
/*
images = [
new Ex21Gif().bitmapData,
new Ex22Gif().bitmapData,
new Ex23Gif().bitmapData,
new Ex24Gif().bitmapData,
new Ex25Gif().bitmapData
];
*/
//***** Flash *****
images = [
new Ex21Gif(0,0),
171
new Ex22Gif(0,0),
new Ex23Gif(0,0),
new Ex24Gif(0,0),
new Ex25Gif(0,0)
];
setNextImage();
frameCounter=0;
finished=false;
}
public function setNextImage():void{
    currentIndex++;
    if (currentIndex > images.length-1) {
        finished=true;
    } else {
        image.bitmapData = images[currentIndex];
    }
}
public function update():void {
    frameCounter++;
}
public function render():void {
    if (frameCounter >= frameDelay && !finished) {
        setNextImage();
        frameCounter=0;
    }
}
```



```
    }  
  }  
  public function dispose():void {  
    removeChild(image);  
    for each ( var tempImage:BitmapData in images ) {  
      tempImage.dispose();  
    }  
    images = null;  
  }  
}  
}
```

## 总结

在这一章，你学习了一些设计游戏的基本概念，设置游戏难度，创建和添加声音、图像，了解 Flash 和 Flex 库的不同支持，使用两个不同的向量算法移动 Sprite，在代码中制作动画。在下一章，我们就会用这些来做我们的第四个游戏----Flak Cannon，下一个计划就是利用我们的游戏框架。



## 第五章

# 构建御空加农炮游戏循环

---

在第四章，我们已经构建好 Flak Cannon 游戏中需要用到的类和素材。在这一章，我们会把这些都有机地组合在一起。概括点说，Flak Cannon 是非常像经典的 Atari Missile Command。你的任务就是守卫你的船队，而你的船队会受到敌人（敢死队飞机）的攻击。你是用一支高射炮来进行防御，玩家不是利用炮弹直接击毁敌人，而是要估算敌人将要到达的位置，利用炮弹的爆炸效果摧毁敌人。

在之前那一章，我们已经创建了一套游戏中要用到的类和对象了。我们再看一下创建了什么吧。SoundManager 类是为了在游戏中能够方便地播放声音。同时，也在游戏中创建了自定义的时间类 CustomEventSound。Shot 类是用来描述高射炮发射的炮弹。我们利用一个点到点的向量来计算炮弹飞行的路径。Ship 和 BonusPlane 用来描述船只和奖励飞机。我们也创建了 Enemy 类来创建敌军飞机，利用



角度和速度来计算飞行路径。还有两个动画类 Flak 和 Explosion。现在，时候利用他们去制作一个完整的游戏了。

## 理解 Flak Cannon 游戏流程

在我们构建这个游戏之前，我们先来看看这个游戏循环的流程和我们需要做什么来让游戏运行起来。

Main.as 这个类文件是游戏的开始入口。它包含新的游戏类 FlakCannon.as，其中，FlakCannon.as 有下面这些函数。

- newGame(): 一个新游戏启动的时候由 Main 调用
- newLevel(): 设置游戏难度的时候由 Main 调用

设置游戏难度: 设置一个新的难度

创建十字瞄准器: 创建十字瞄准器并放置在屏幕上

- placeShips(): 在屏幕上摆放船队
- runGame(): 在游戏循环中被 Main 调用
- checkEnemies(): 检查是否已经创建一个敌军飞机
- checkBonusPlane(): 检查是否已经创建一个奖励飞机
- update(): 计算所有对象的新位置
- checkCollisions(): 检查碰撞
- addToScore(): 加分
- render(): 渲染器
- checkforEndLevel(): 检查是否过关了
- addToScore(): 根据剩余的炮弹加分
- cleanUpLevel(): 清理所有对象
- checkforEndGame(): 检查玩家是否已经输了
- shootListener(): 监听鼠标单击事件（鼠标单击就发炮）
- mouseMoveListener(): 监听鼠标移动事件（更新十字瞄准器的位置）



## 更新 GameFramework.as

在我们接触 Flak Cannon 的核心代码之前, 我们需要讨论一下支持 SoundManager 的一些游戏框架更新。这些改变不是针对 Flak Cannon 这个游戏的, 而是针对所有的游戏框架。我们在这章里面放置一些代码, 使得作为游戏框架中一部分的 SoundManager 能够独立运行。这章会告诉我们应该怎么样把它整合到其余的代码中去。

在 GameFrameWork.as 中首先要做的事情就是, 声明一个变量来持有 SoundManager 的引用。我们把这个变量命名为 soundManager, 并把他添加为 GameFrameWork.as 的成员变量。

```
public var soundManager:SoundManager;
```

**注意：**这时候建议你在书的背面上的网址中下载源码。你可以在 `/source/class/com/efg/framework/` 中找到最新版本的 GameFrameWork.as。如果你回想起第四章的 CustomSoundEvent, 是否还记得他就是用来告诉 Main.as 游戏中需要播放一个声音。当我们需要使用这个事件, 我们需要两个函数。第一个是 systemNewGame()。我们需要为 GameFrameWork 添加 EventListener 的功能。第二个函数是 soundEventListener(), 用来监听抛出的事件。

## 构建 FLAK CANNON 游戏循环

```
14. public function systemNewGame():void {  
15.     addChild(game);  
16.     game.addEventListener(CustomEventScoreBoardUpdate.UPDATE_TEXT,  
scoreBoardUpdateListener, false, 0, true);  
17.     game.addEventListener(CustomEventLevelScreenUpdate.UPDATE_TEXT,  
levelScreenUpdateListener, false, 0, true);  
18.     game.addEventListener(CustomEventSound.PLAY_SOUND, soundEventListener, false, 0,  
true);  
19.     game.addEventListener(Game.GAME_OVER, gameOverListener, false, 0, true);  
20.     game.addEventListener(Game.NEW_LEVEL, newLevelListener, false, 0, true);  
21.     game.newGame();  
22.     switchSystemState(FrameWorkStates.STATE_SYSTEM_NEW_LEVEL);  
}
```

估计很多的开发人员喜欢用 onXxxxxEvent() 这样的名字去命名事件的回调函数。如果你需要保持这种风格的话, 你可以重命名这些回调函数的名字。当游戏结束的时候, 我们需要把这些事件监听去掉。

```
1. public function gameOverListener(e:Event):void {
```



```

2.  switchSystemState(FrameWorkStates.STATE_SYSTEM_GAME_OVER);
3.  game.removeEventListener(CustomEventScoreBoardUpdate.UPDATE_TEXT,
scoreBoardUpdateListener);
4.  game.removeEventListener(CustomEventLevelScreenUpdate.UPDATE_TEXT,
levelScreenUpdateListener);
5.  game.removeEventListener(CustomEventSound.PLAY_SOUND, soundEventListener);
6.  game.removeEventListener(Game.GAME_OVER, gameOverListener);
7.  game.removeEventListener(Game.NEW_LEVEL, newLevelListener);
8.  }

```

soundEventListener()是在 GameFrameWork 类中声明的.当 CustomEventSound 的事件被抛出,我们会设置 CustomEventSound 的属性来播放声音,先回顾一下有哪些属性:

type:这个字符串用来定义事件类型,例如: CustomSoundevent.PLAY\_SOUND 和 CustomSoundevent.STOP\_SOUND

name:这个字符串表示播放哪个声音,在 Main.as 中定义,如: Main.SOUND\_XXX

isSoundTrack:这个布尔变量用来判断声音是否为声道

**注意:**我们会在第八章讨论声道的相关知识

loops:这个整形变量用来记录一个声音循环播放多少次。

offset : 用来定义声音之间的间隔,以毫秒为单位。

volume : 音量,大小从 0 到 1。

当 soundEventListener()函数被调用,我们通过观察 type 参数来决定播放还是停止一个声音。除此之外,所有的参数都会传入到 soundManager.playSound()中。

```

1.  public function soundEventListener(e:CustomEventSound):void {
2.  if (e.type == CustomEventSound.PLAY_SOUND) {
3.  soundManager.playSound(e.name, e.isSoundTrack, e.loops, e.offset, e.volume );
4.  }else {
5.  soundManager.stopSound(e.name, e.isSoundTrack);
6.  }
7.  }

```

## 定义 Main.as

现在,我们来定义 Flak Cannon 的 Main.as。我们已经把游戏框架的大部分抽象出来了。这样做可以



更方便的定义我们的游戏，即使是用到其他的游戏，都可以方便重用。(除非你要添加新的特性).Main.as 中需要添加一些 Import 语句，尤其要注意添加我们新创建的 SoundManager。

```
1. package com.efg.games.flakcannon
2. {
3.     import com.efg.framework.FrameWorkStates;
4.     import com.efg.framework.GameFrameWork;
5.     import com.efg.framework.BasicScreen;
6.     import com.efg.framework.ScoreBoard;
7.     import com.efg.framework.Game;
8.     import com.efg.framework.SideBySideScoreElement;
9.     import com.efg.framework.SoundManager;
10.    import flash.display.Bitmap;
11.    import flash.display.BitmapData;
12.    import flash.display.Sprite;
13.    import flash.events.Event;
14.    import flash.geom.Point;
15.    import flash.utils.Timer;
16.    import flash.events.TimerEvent;
17.    import flash.text.TextFormat;
```

Main 类是继承 com.efg.framework.GameFrameWork 的，意味着我们只需要去覆盖一些我们需要改变的方法。在 Flak Cannon 中，需要覆盖的是 init()。

```
1. public class Main extends GameFrameWork {
```

//自定义用来加分的元素

这些元素和上一个游戏不同，不过原理是一样的。

```
1. public static const SCORE_BOARD_SCORE:String = "score";
2. public static const SCORE_BOARD_SHOTS:String = "shots";
3. public static const SCORE_BOARD_SHIPS:String = "ships";
```

对于 Flak Cannon，能够播放声音是很有意义的。我们通过创建一些静态常量来关联游戏中的声音。



//自定义不同的声音

```
1. public static const SOUND_BONUS:String = "sound bonus";
2. public static const SOUND_BONUS_SHIP:String = "sound bonus ship";
3. public static const SOUND_SHOOT:String = "sound shoot";
4. public static const SOUND_NOSHOTS:String = "sound no shots";
5. public static const SOUND_EXPLODE_PLANE:String = "sound explode plane";
6. public static const SOUND_EXPLODE_FLAK:String = "sound explode flak";
7. public static const SOUND_EXPLODE_SHIP:String = "sound explode ship";
```

/\*\*Flex 程序专用

```
1. /*
2. [Embed(source = "assets/flakassets.swf", symbol="SoundExplodePlane")]
3. private var SoundExplodePlane:Class;
4. [Embed(source = "assets/flakassets.swf", symbol="SoundExplodeFlak")]
5. private var SoundExplodeFlak:Class;
6. [Embed(source = "assets/flakassets.swf", symbol="SoundShoot")]
7. private var SoundShoot:Class;
8. [Embed(source = "assets/flakassets.swf", symbol="SoundNoShots")]
9. private var SoundNoShots:Class;
10. [Embed(source = "assets/flakassets.swf", symbol="SoundBonus")]
11. private var SoundBonus:Class;
12. [Embed(source = "assets/flakassets.swf", symbol="SoundBonusShip")]
13. private var SoundBonusShip:Class;
14. [Embed(source = "assets/flakassets.swf", symbol="SoundExplodeShip")]
15. private var SoundExplodeShip:Class;
16. */
17. public function Main() {
18. init();
19. }
```

我们要覆盖的函数就只有 init()，覆盖的内容大多都是包括 Flak Cannon 的特性。游戏框架的其他部分不用改。第一个改动，我们会传递 width 和 height 这两个参数到 Flak Cannon 的实例。Flak Cannon 的实例会利用这两个值去设置游戏的边界大小。

另外一些比较突出的改变：大小，位置，文字更新。游戏的界面大小是 600\*400 个像素，还需要一个



计分板来记录分数，弹药量，船只数量。我们在屏幕上添加 “Flak Cannon Go! and Shoot The Enemy Planes!” 这个文字说明。还要添加一个 “Play!” 按钮，在屏幕下方再加上一个 “Play Again” 按钮。

```
1.  override public function init():void {
2.  game = new FlakCannon(600,400);
3.  setApplicationBackGround(600,400,false, 0x0042AD);
4.  scoreBoard = new ScoreBoard();
5.  addChild(scoreBoard);
6.  scoreBoardTextFormat= new TextFormat("_sans", "11", "0xffffffff", "true");
7.  scoreBoard.createTextElement(SCORE_BOARD_SCORE, new SideBySideScoreElement(80,
    5, 15, "Score",scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));
8.  scoreBoard.createTextElement(SCORE_BOARD_SHOTS, new
    SideBySideScoreElement(240, 5, 10, "Shots",scoreBoardTextFormat, 40, "0",
    scoreBoardTextFormat));
9.  scoreBoard.createTextElement(SCORE_BOARD_SHIPS, new SideBySideScoreElement(400,
    5, 10, "Ships",scoreBoardTextFormat, 50, "0", scoreBoardTextFormat));
10. screenTextFormat = new TextFormat("_sans", "14", "0xffffffff", "true");
11. screenButtonFormat = new TextFormat("_sans", "11", "0x000000", "true");
12. titleScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE, 600,400, false,
    0x0042AD);
13. titleScreen.createDisplayText("Flak Cannon",250,new Point(255,100), screenTextFormat);
14. titleScreen.createOkButton("Go!",new Point(250,250),100,20,
    screenButtonFormat,0xFFFFFFFF,0xFF0000,2);
15. instructionsScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS,
    600,400, false, 0x0042AD);
16. instructionsScreen.createDisplayText("Shoot The Enemy Planes!",300, new
    Point(210,100),screenTextFormat);
17. instructionsScreen.createOkButton("Play",new Point(250,250),100,20,
    screenButtonFormat,0xFFFFFFFF,0xFF0000,2);
18. gameOverScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER,
    600,400, false, 0x0042AD);
19. gameOverScreen.createDisplayText("Game Over",300,new Point(250,100),
    screenTextFormat);
20. gameOverScreen.createOkButton("Play Again", new Point(225,250),150,20,
    screenButtonFormat,0xFFFFFFFF,0xFF0000,2);
21. levelInText = "Level ";
22. levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER,
    600,400, false, 0x0042AD);
23. levelInScreen.createDisplayText(levelInText,300,new Point(275,100), screenTextFormat);
```



```
24. switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
25. waitTime= 40;
```

接下来的代码我们需要思考一下：我们建立了一些用来表达声音的常量（`public static const SOUND_BONUS:String = "sound bonus";`），怎么样才能把常量和声音关联起来呢？我们需要把两个参数传入到 `soundManager.addSound()` 中，一个是表示声音的常量字符串，另一个是声音对象的引用。

```
1. soundManager.addSound(SOUND_BONUS,new SoundBonus());
2. soundManager.addSound(SOUND_BONUS_SHIP, new SoundBonusShip());
3. soundManager.addSound(SOUND_SHOOT,new SoundShoot());
4. soundManager.addSound(SOUND_NOSHOTS,new SoundNoShots());
5. soundManager.addSound(SOUND_EXPLODE_PLANE,new SoundExplodePlane());
6. soundManager.addSound(SOUND_EXPLODE_FLAK,new SoundExplodeFlak());
7. soundManager.addSound(SOUND_EXPLODE_SHIP,new SoundExplodeShip());
8. frameRate = 30;
9. startTimer();
10. }
11. }
12. }
```

## FlakCannon.as

每一个游戏都会有一个继承自 `Game` 类的独一无二的类。在这个游戏里，这个类就是 `FlakCannon`。这一章的大部分内容都是围绕 `FlakCannon.as` 的编写。

## 导入类

第一件事，就是导入游戏中需要用到的类。其中有些你已经在 `Super Click` 里见过了。

```
1. package com.efg.games.flakcannon {
2. import flash.display.Sprite;
3. import flash.events.*;
4. import flash.events.MouseEvent;
```

我们会用鼠标去控制十字瞄准器，因此，我们必须导入 `MouseEvent` 类。剩下的导入的类都是之前



没见过的。不过在这一节里面会介绍到如何使用这些类的。

首先，我们会把游戏中要用到的图片导进来，这个过程我们需要 Bitmap 来支持这个处理过程。

```
1. import flash.display.Bitmap;
```

下一步，我们用 BitmapData 对象来做位图级别的碰撞检测。同时，也需要一个 Point(x,y)去检查碰撞。

```
1. import flash.geom.Point;
```

我们需要知道鼠标在屏幕上的位置，这样我们才能知道发射炮弹的目的地。ui.Mouse 可以提供这些信息。

```
1. import flash.ui.Mouse;
```

最后，我们导入游戏中需要用到的其他类

```
1. import com.efg.framework.Game;
2. import com.efg.framework.CustomEventLevelScreenUpdate;
3. import com.efg.framework.CustomEventScoreBoardUpdate;
4. import com.efg.framework.CustomEventSound;
5. public class FlakCannon extends com.efg.framework.Game {
```

## 设置 FlakCannon 类的成员属性和构造方法

现在，我们已经把相关的导入语句写好了，是时候来点实际的了。首先，我们必须定义类中的成员属性，开头的两个属性，Width and gameHeight，用来存储游戏界面的宽和高。

//构造函数

```
1. private var gameWidth:int;
2. private var gameHeight:int;
```

接下来，还会定义一些属性来记录 Flak Cannon 的信息。score 用来记录玩家的分数.level 用来记录玩家当前所在的关卡数，这是用来创建难度设定的。ships 是用来记录当前玩家拥有的船只数量。shots 记录弹药量。游戏会通过 CustomEventScoreBoardUpdate 事件来刷新这些信息，这些信息都显示在计分板面上。isGameOver 变量用来判断玩家是否已经输了。extraCount 用来记录玩家额外得到的船只。

```
1. private var score:int;
2. private var level:int;
3. private var ships:int;
4. private var shots:int;
5. private var isGameOver:Boolean = false;
6. private var extraCount:int;
```

下一个变量时用来记录玩家获得了多少个额外的船只（要得到额外的船只，玩家获取的分数必须到达 scoreNeededForExtraShip 这个值）。由于玩家很难说刚刚好到达 scoreNeededForExtraShip 这个值，



所以我们需要保存玩家已经得到的额外分数。

```
1. private var extraShipCount:int = 0;
```

现在，我们会创建数组和变量来存储游戏中的对象。Explosion 存储在 explodeArray，Shot 存储在 shotArray，Ship 存储在 shipArray，Flak 的爆炸效果存储在 flakArray，Enemy 存储在 enemyArray，BonusPlane 存储在 bonusPlaneArray，crosshairs 用来存储十字瞄准器。

```
1. private var explodeArray:Array;  
2. private var shotArray:Array;  
3. private var shipArray:Array;  
4. private var enemyArray:Array;  
5. private var flakArray:Array;  
6. private var bonusPlaneArray:Array;  
7. private var crosshairs:Bitmap;
```

incomingCount 表明在一关里面创建的敌军飞机数量

我们通过 incomingCount 的值来判断这一关是否已经结束。

```
private var incomingCount:int;
```

下面的所有数据在 FlakCannon 中都要用到，所以我们把这些数据设为这个类的成员变量。这些数据是用来控制游戏的难度的。你可以在第四章看到详细的描述。

```
1. private var enemyWaveDelay:int = 30;  
2. private var numEnemies:int;  
3. private var enemyFrameCounter:int = 0;  
4. private var enemySpeed:int = 0;  
5. private var enemyWaveMax:Number = 0  
6. private var enemyWaveMultipleChance:Number = 0;  
7. private var enemySideFloor:Number = 100;  
8. private var enemySideChance:Number = 10;  
9. private var bonusPlaneFrameCounter:Number = 0;  
10. private var bonusPlaneDelay:Number = 1;  
11. private var startShots:int = 30;  
12. private var scoreNeededForExtraShip:int = 10000;  
13. private var baseEnemyScore:int = 250;  
14. private var enemyHitBonus:int = 500;  
15. private var baseBonusPlaneScore:int = 500;  
16. private var maxBonusPlanes:int = 1;  
17. private var maxVisibleShips:int = 3;
```



```
18. private var bonusPlaneMaxY:int =350;
19. private var bonusPlaneSpeed:int = 3;
20. private var shipYPadding:int = 0;
```

下面我们定义了一些变量，这些变量是用来临时存储某些信息的。它们是用来保存游戏中主要对象的实例 ( Shot, Flak, Enemy, Ship, Explosion, BonusPlane )。

```
1. private var tempShot:Shot;
2. private var tempFlak:Flak;
3. private var tempEnemy:Enemy;
4. private var tempShip:Ship;
5. private var tempExplode:Explosion;
6. private var tempBonusPlane:BonusPlane;
```

另外，我们还给 flex 版本准备了 flakassets.swf。

```
1. [Embed(source = "assets/flakassets.swf", symbol="CrosshairsGif")]
2. private var CrosshairsGif:Class;
```

接下来，我们定义构造函数。调用这个函数只是用来设置 gameWidth 和 gameHeight，不做其他事了。等一下，我们会利用 gameWidth 和 gameHeight 去计算一个很重要的值。FlakCannon 游戏现在就是等待 Main 类去调用它的函数，第一个就是 newGame()。

```
1. override public function newGame():void {
2.   level = 0 ;
3.   score = 0;
4.   ships = 3;
5.   shots = 0;
6.   extraShipCount=0;
7.   isGameOver = false;
8.   dispatchEvent(new
      CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC
      ORE_BOARD_SCORE,"0"));
9.   dispatchEvent(new
      CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC
      ORE_BOARD_SHOTS,String(shots)));
10. }
```

newGame()中的代码非常简单，和 Super Click 比起来也就是两个小小的改动。第一，我们初始化了与游戏基本信息相关的成员变量。

```
1. level = 0 ;
2. score = 0;
```



```
3. ships = 3;
4. shots = 0;
5. extraShipCount=0;
6. isGameOver = false;
```

接下来，我们定义了一些事件来告诉 Main 类玩家在游戏中的相关信息。

```
1. dispatchEvent(new
   CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC
   ORE_BOARD_SCORE,"0"));
2. dispatchEvent(new
   CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC
   ORE_BOARD_SHOTS,String(shots)));
```

/\*\*\*\*\*第二部分开始\*\*\*\*\*/

我们在 Super Click 中也有创建类似的事件。这里的不同之处是：我们会利用 CustomEventScoreBoard 事件发送玩家剩余的船只数量到 ScoreBoard.

## 开始下一关

newLevel()是我们更新游戏关卡设置的地方。在改变这些设置之前，我们必须先把所有用来保存游戏相关对象的数组初始化。同时也需要重置 incomingCount 和更新 level 变量。

```
1. override public function newLevel():void {
2. explodeArray = [];
3. flakArray = [];
4. shotArray = [];
5. shipArray = [];
6. enemyArray = [];
7. bonusPlaneArray = [];
8. incomingCount = 0;
9. level++;
```

下面是第四章讲到的难度设置。上一章已经有怎样使用这些变量的详细说明了。我们也有在 FlakCannon 类中的各个方法中讨论过这些设置的用法。现在，只需要注意，我们可以通过改变这些设置来改变游戏的玩法。



```
1. numEnemies = (numEnemies > 100) ? 100 : level * 10 + (level*5);
2. enemyWaveDelay = (enemyWaveDelay < 20) ? 20 : 60 - (level-1)*2;
3. enemyWaveMax = (enemyWaveMax > 8) ? 8 : 1 * level+1;
4. enemyWaveMultipleChance = (enemyWaveMultipleChance > 100) ? 100 : 10*level;
5. enemySpeed = (enemySpeed > 8) ? 8 : 2 + (.5*(level-1));
6. enemySideChance = (enemySideChance > 70) ? 70 : 10*(level-1);
7. enemySideFloor = (enemySideFloor > 300) ? 300 : 100 + 25*(level-1);
8. bonusPlaneDelay = (bonusPlaneDelay > 450) ? 450 : 350 + 10*(level-1);
9. bonusPlaneSpeed = (bonusPlaneSpeed > 12) ? 12 : 4 + (1*level);
10. bonusPlaneFrameCounter = 0;
11. enemyFrameCounter = enemyWaveDelay;
12. startShots = 30;
13. shots+=startShots;
14. scoreNeededForExtraShip = 10000;
15. baseEnemyScore = 100;
16. enemyHitBonus = 500;
17. baseBonusPlaneScore = 500;
18. maxBonusPlanes = 1;
19. maxVisibleShips = 3;
20. bonusPlaneMaxY = 350;
21. shipYPadding = 5;
```

## 处理鼠标事件和十字瞄准器

玩家在屏幕上控制十字瞄准器来确定射击目标。点击鼠标就会发射炮弹。为了可以使用十字瞄准器，我们必须要在 Game 类中定义一个成员变量来保存这个 bitmap 对象。不过，我们之前已经声明了这个变量了。

```
1. private var crosshairs:Bitmap;
```

现在，我们会在 newLevel() 中测试十字瞄准器是否存在。如果不存在，就用 Mouse 类的 hide() 函数把鼠标指针隐藏掉，然后再用 crosshairs 去装载十字瞄准器的位图并添加到屏幕上。注意：如果你是使用 Flex 的，创建一个新实例的调用和 Flash 是不同的。这是因为 Flex 把 CrosshairsGif() 弄成一个 Bitmap 了，而 Flash 需要传入 CrosshairsGif() 的实例去重新创建一个新的 Bitmap。区别很细微。两部分的代码都是很必要的。

```
1. if (crosshairs == null) {
2.     Mouse.hide();
3.     //***** Flex *****
4.     //crosshairs = new CrosshairsGif();
```



```
5. //***** Flash *****
6. crosshairs = new Bitmap(new CrosshairsGif(0,0));
```

现在，我们需要把十字瞄准器添加到屏幕上。这是我们第一次用到从 Main 从传递过来的 `gameWidth` 和 `gameHeight`，我们会用 `addChild()` 方法把十字瞄准器添加到屏幕上。最后，我们要在舞台上添加一个处理函数来监听 `MouseEvent.MOUSE_MOVE` 事件，这样一来，就可以通过鼠标来移动十字瞄准器。

```
1. crosshairs.x=gameWidth/2;
2. crosshairs.y=gameHeight/2;
3. addChild(crosshairs);
4. stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener, false, 0, true);
5. }
```

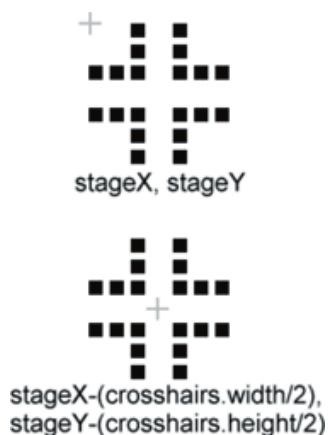
现在我们已经把十字瞄准器添加到屏幕上了，不过我们还需要支持用户输入。我们还需要监听鼠标 `MouseEvent.MOUSE_DOWN` 事件，这样我们才知道什么时候发射炮弹。在这个游戏里，`MouseEvent.MOUSE_DOWN` 事件会比 `MouseEvent.MOUSE_CLICK` 来得好，这是因为 `MouseEvent.MOUSE_CLICK` 需要按下鼠标并释放鼠标才抛出事件，而 `MouseEvent.MOUSE_DOWN` 只需要按下鼠标就会触发事件了。这样的体验会更好。

```
1. stage.addEventListener(MouseEvent.MOUSE_DOWN, shootListener, false, 0, true);
```

我们已经给事件添加了监听，也是时候创建响应函数了。第一个函数是 `mouseMoveListener()`，十分简单，当鼠标移动的时候，它就会更新十字瞄准器的位置。

```
1. public function mouseMoveListener(e:MouseEvent):void {
2.   crosshairs.x = e.stageX-(crosshairs.width/2);
3.   crosshairs.y = e.stageY-(crosshairs.height/2);
4. }
```

这个函数有两个地方值得注意一下。第一个非常明显但仍需注意。我们可以看到，鼠标的位置和使用 `MouseEvent` 对象的 `stageX` 和 `stageY` 属性的舞台相关。第二个不同点就更细微了。我们在放置十字瞄准器的时候需要从 `stageX` 和 `stageY` 中减去十字瞄准器的宽和高的一半。看图【5-1】





图【5-1】把十字瞄准器调整到中间。

我们这样做事因为在 AS3 中，bitmaps 是动态加载的。如果没有任何调整，一个 bitmap 图片会显示在注册点（看回之前的图【4-12】）。如果不这么做的话，当发射炮弹之后，炮弹就会在十字瞄准器的左上角爆炸，而不是中间了。

调整之后，你就能够看到十字瞄准器的中间就是鼠标所在的地方和炮弹的目的地了。接下来，需要写一个函数来监听玩家点击鼠标来发射炮弹。

```
1. public function shootListener(e:MouseEvent):void {
```

在这个函数里面首先就是检查一下玩家还又没有炮弹剩余。很简单，一个 if 语句：如果炮弹还有剩，那就执行 if 里面的语句，创建一个 Shot 类的实例，然后添加到舞台上，并把之放入到 shotArray 中。接着弹药量 shots 减 1，mouseX-(crosshairs.width/2),mouseY-(crosshairs.height/2)这一句是为了让在十字瞄准器的中间爆炸。

```
1. if (shots > 0) {
2.   tempShot = new
     Shot(gameWidth/2,gameHeight,mouseX-(crosshairs.width/2),mouseY-(crosshairs.height
       /2));
3.   this.addChild(tempShot);
4.   shotArray.push(tempShot);
5.   shots--;
6.   dispatchEvent(new
     CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,
       Main.SCORE_BOARD_SHOTS,String(shots)));
7.   dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND,
     Main.SOUND_SHOOT,false,1,0,.5));
```

最后，我们发送一个消息到计分板 ScoreBoard，告诉它弹药量 shots 的值变了，让它更新 shots 的值。同时，也抛出一个 CustomSoundEvent 事件去播放一个发射炮弹的声音。

当没有弹药的时候，我们就不会执行 if 语句了，而是执行下面的 else 语句。主要是播放一个没有弹药的声音，好让玩家知道弹药已经用完了。

```
8.   } else {
9.     dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND,
       Main.SOUND_NOSHOTS,false,1,0,.75));
10.  }
11. }
```



## 放置船队（船只）

`newLevel()`中最后要做的事情就是把玩家守护的船队添加到屏幕上。在每一关开始的时候，游戏都会重置一次船队，这样，当船队中船只的数量发生变化，我们都可以正确的把它显示在屏幕上。不管当前玩家的船队有多少船只，任何时候屏幕上面只是显示 3 只船。这个是由 `maxVisibleShips` 控制的。如果少于 3 只船，就调整位置，让船队居中。

要处理船队的位置摆放，就要调用 `newLevel()`中的 `placeShips()`。

### 1. `placeShips()`;

现在，我们需要定义 `placeShips()`了。这个函数的功能就是把船队总是放在屏幕的中间位置。为了达成这个目的，需要知道玩家有多少船只和屏幕的大小。很幸运，我们已经知道这些值了，我们只是需要使用就好了。第一件事就是定义函数和找出玩家还有多少只船。

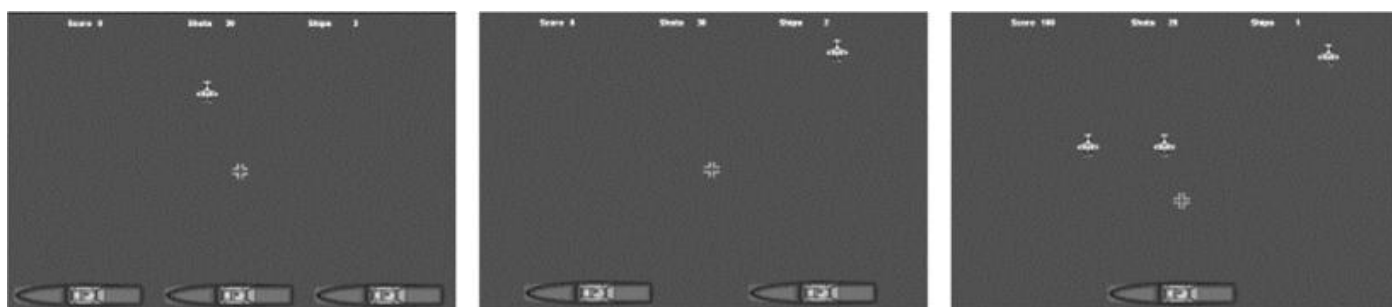
```
1. private function placeShips():void {  
2.   var ctr:int;  
3.   var xSpacing:int;  
4.   var tShips:int = ships;  
5.   if (tShips > maxVisibleShips) {  
6.     tShips = maxVisibleShips;  
7.   }
```

我们并不关心在每一关的开始玩家拥有的船只数量是否大于 3；我们只是要 `maxVisibleShips` 这个值。多出来的船只只会存储起来。现在，我们开始放置船队了。

`xSpacing = (gameWidth/tShips);`（屏幕被分成 `tShips` 部分）

`xSpacing` 是由屏幕宽度除以玩家拥有船只数量的出来的，我们会在第二步中用到这个值。下面我们利用一个循环来设置好船只的位置。

```
1. for (ctr = 0; ctr < tShips; ctr++) {  
2.   tempShip = new Ship();  
3.   tempShip.x = ((xSpacing * (ctr+1)) - xSpacing/2) - (tempShip.width/2); //判断当前船只所在  
   屏幕的哪一部分，并把船只放到该部分的中间  
4.   tempShip.y = gameHeight-tempShip.height-  
   shipYPadding; //gameHeight-tempShip.height 是一个固定的值，shipYPadding 越大，则游  
   戏难度越高，因为 shipYPadding 越大，船队就越靠近屏幕的上方（而敌军是从屏幕上方出现的）  
5.   shipArray.push(tempShip); //把这个临时变量放到 shipArray 中，以后则可以通过 shipArray 来  
   控制这些船只的摆放。  
6.   this.addChild(tempShip); //添加到屏幕上  
7. }  
8. }
```



图【5-2】放置船只

## 处理新的一关开始事件

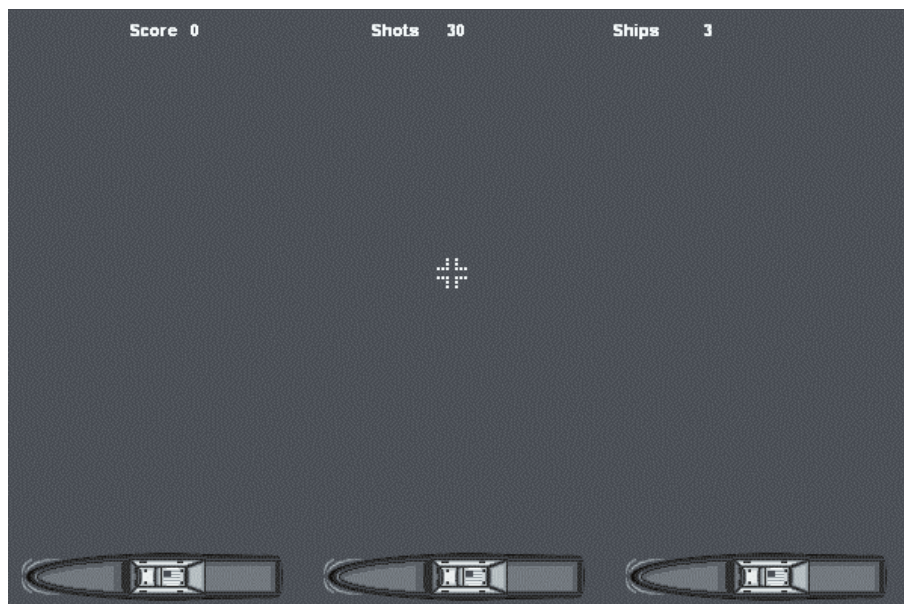
现在，我们已经几乎完成了 `newLevel()`。剩下要做的只是通过抛出事件，把 `level`, `shots`, `ships` 这些数据通知到计分板，并更新显示这些数据。

1. `dispatchEvent(new CustomEventLevelScreenUpdate(CustomEventLevelScreenUpdate.UPDATE_TEXT,String(level)));`
2. `dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_SHOTS,String(shots)));`
3. `dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_SHIPS,String(ships)));`

## 检验游戏

重述一下上面说到的，我们创建了 4 个新的函数：`newLevel()`, `mouseMoveListener`, `shootListener` 和 `placeShips()`。如果你现在编译并运行游戏，你会看到类似于图【5-3】的情形。如果你点击鼠标，你会听到发射炮弹的声音，但是不会看到有炮弹发射，那是因为那部分的代码还没有写好。

你也可以通过移动鼠标来移动十字瞄准器。如果你完成了，就代表我们可以成功创建一个关卡了。



图【5-3】

## 创建游戏循环

最后，是时候说回游戏本身的架构了。和 `newGame()`，`newLevel()` 一样，`runGame()` 也是作为游戏框架的一部分被 `Main` 调用的。游戏框架最好的地方就在于我们不需要担心 `runGame()` 什么时候，以什么方式被调用。

我们只需要知道 `runGame()` 的细节。我们先来看看函数的内容是怎样的。

```
1. override public function runGame():void {  
2.   checkEnemies();  
3.   checkBonusPlane();  
4.   update();  
5.   checkCollisions();  
6.   render();  
7.   checkforEndLevel();  
8.   checkforEndGame();  
9. }
```

非常简单。它只是简单的调用一群函数而已，不过这群函数也很容易实现。在下一节，我们会讨论这些函数的实现，和这些函数是怎样结合起来的。



## 检查敌军

在 `runGame()` 中第一个函数就是 `checkEnemies()`。这个函数会用到 `enemyWaveDelay` 和 `enemyFrameCounter`，功能就是判断是否要创建一个敌机去袭击玩家的船队。前面两行代码用来判断是否需要创建一个敌军。首先先给 `enemyFrameCounter` 增加 1，再和 `enemyWaveDelay` 做比较。在第一关，`enemyWaveDelay` 的值是 60，这意味着在创建一个敌机之前要消耗 60 帧。也要比较 `incomingCount` 和 `numEnemies`。每创建一个敌军，`incomingCount` 都会增加。`numEnemies` 是设置每一关里面敌军数量的。在第一关里面，它的值是 15。

```
1. private function checkEnemies():void {  
2.     enemyFrameCounter++;  
3.     if ((enemyFrameCounter >= enemyWaveDelay) && (incomingCount < numEnemies))  
4.     {  
5.         var chance:int = Math.floor(Math.random() * 100)+1;  
6.         var enemiesToCreate:int = 1;
```

现在，我们需要判断是否要创建一波多架敌军的攻击，如果是的话，我们要创建多少个敌机。这里，我们用 `enemyWaveMultipleChance` 和一个 1 到 100 的随机数 `chance` 作比较。如果 `chance` 小于等于 `enemyWaveMultipleChance`，那么我们就创建一波敌机，而不是一架。一次创建的敌机越多，则难度越高。

```
7.         var chance:int = Math.floor(Math.random() * 100)+1;  
8.         var enemiesToCreate:int = 1;
```

图【5-4】展示敌机以一定的时间间隔出现的情形。



图【5-4】

如果真的是一波多架敌机的攻击（如图【5-5】），那就必须知道最多随机创建了多少个敌机，这个上限，由 `enemyWaveMax` 决定。在这个函数里，`enemiesToCreate` 就是创建敌机数量的值。

```
1.   if (chance <= enemyWaveMultipleChance)
2.   {
3.       enemiesToCreate = Math.floor(Math.random() * enemyWaveMax)+1;
4.   }
```



图【5-5】一波多架敌机的攻击

我们要判断一下 `enemiesToCreate` 是否会大于还没有出场的敌机数量。



numEnemies-incomingCount 的值就是总的敌军数量减去已经创建敌军数量, enemiesToCreate 是不能大于这个值的。

```
1.     if (enemiesToCreate > (numEnemies-incomingCount))
2.     {
3.         enemiesToCreate = (numEnemies-incomingCount);
4.     }
```

现在, 是时候创建敌机并添加到舞台上了。我们会利用一个 for 循环来做, 以 enemiesToCreate 作为循环次数的上限。在第四章, 我们定义的 Enemy 类需要一个始点和终点来构造一个向量, 这个向量会决定敌机飞行的方向。一开始, 我们先把 startX, startY, endX, endY, dir 初始化吧。

```
1.     for (var ctr:int = 0; ctr < enemiesToCreate; ctr++) {
2.         var startX:int = 0;
3.         var startY:int = 0;
4.         var endX:int = 0;
5.         var endY:int = 0;
6.         var dir:int = 0;
7.         chance = Math.floor(Math.random() * 100)+1;
```

我们在这里会用到另外一个设置: enemySideChance。用来确定这个敌机的飞行方向的。确定的方法和上面确定一波多架敌机攻击中的敌机数量很相似, 都要用到一个随机数。如果这个敌机真的是从边路出现的, 之后还要再用一个随机数来判断是从左边还是右边出现。

```
8.         if (chance <= enemySideChance)
9.         {
10.            var leftOrRight:int = Math.floor(Math.random() * 2);
```

startY 是敌机始点的 y 坐标, 我们要也要根据另一个设置: enemySideFloor, 来计算这个坐标值。enemySideFloor 是表示一架敌机从边路出现的最低点。这里的设计意图是这样的, 如果敌机出现的地方越靠近屏幕下方, 则这敌机就越难被打下来。

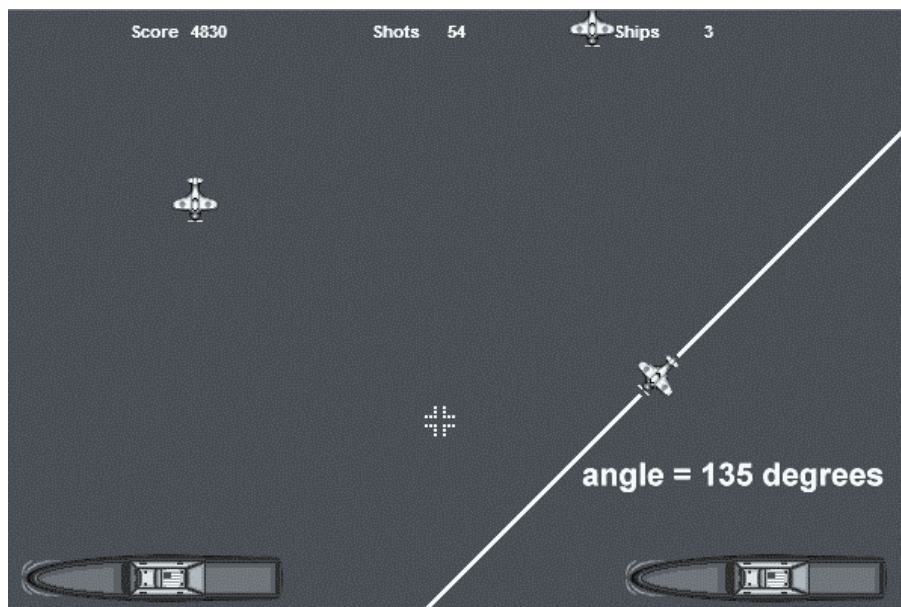
endY 的值就是游戏屏幕的高度 gameHeight, 就是说除非被击毁, 敌机只有到达屏幕的最底部才会消失。我们不需要计算 endX 的值, 因为只需要判断 endY 的值就知道敌机是否到达终点了。接下来, 就要看敌机是从左往右飞还是从右往左飞了。

这两种情况会有些不同。飞向左边的, 敌机的 x 坐标就是屏幕的右边边界, 坐标值就是 gameWidth。飞向右边的敌机的 x 坐标就是 0 再减去飞机本身的宽度。如果你还记得第四章的内容, 敌机飞行需要选择一个角度, 向右飞的角度是 45 度, 向左飞的角度是 135 度。看图【5-6】。

```
11.        startY = Math.floor(Math.random() * enemySideFloor)+1;
12.        endY = gameHeight;
13.        switch(leftOrRight)
14.        {
15.            case 0: //left
```



```
16.     startX = gameWidth;  
17.     dir = Enemy.DIR_LEFT;  
18.     break;  
19.     case 1: //right  
20.     startX = -32;  
21.     dir = Enemy.DIR_RIGHT;  
22.     break;  
23. }
```



图【5-6】。

基本上大部分的敌机都是向下飞的，所以始点和终点的坐标很容易算。始点的 x 坐标就是 0 到游戏屏幕的宽度减去敌机的宽度中的任意一个值。这样可以把敌机限制在屏幕里面。startY 的值就是-32 了，原因你应该懂的。endY 的值就是游戏屏幕的高度了。

```
24.     }  
25.     else  
26.     {  
27.         startX = Math.floor(Math.random() * (gameWidth-32));  
28.         startY = -32;  
29.         endY = gameHeight;  
30.         dir = Enemy.DIR_DOWN;  
31.     }
```

接下来，我们用上面计算出来的值去创建敌机的实例，然后用 addChild() 添加到舞台上，并把引用放到 enemyArray 中。同时也要更新 incomingCount 的值，这样，我们就可以判断这一关是否已经创建好了。

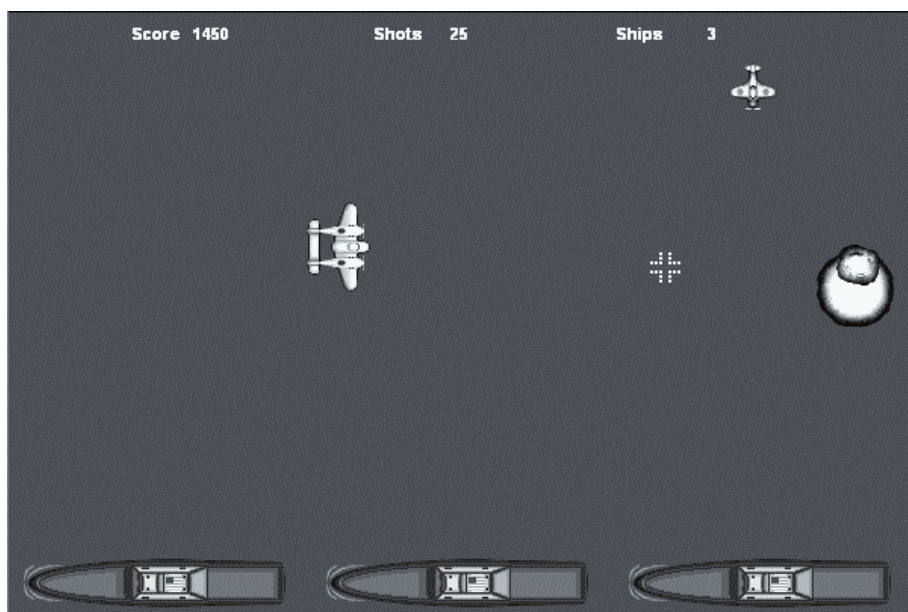


最后，要把 enemyFrameCounter 置 0。

```
32.     tempEnemy = new Enemy(startX,startY,endY,enemySpeed,dir);
33.     this.addChild(tempEnemy);
34.     enemyArray.push(tempEnemy);
35.     incomingCount++;
36. }
37.     enemyFrameCounter = 0;
38. }
39. }
```

## 检验奖励飞机

如图【5-7】所示，一架奖励飞机会在以一定的时间间隔出现，玩家通过击毁它可以获得额外的炮弹。



图【5-7】

下面是奖励飞机和敌机的不同之处：

- 1.奖励飞机总是从左往右飞。
- 2.击中奖励飞机可以给玩家奖励 10 个炮弹。
- 3.出现的奖励飞机数量以 maxBonusPlanes 为上限。

除了这些规则不同，checkBonusPlane()和 CheckEnemies()都是差不多的。一开始，我们用到一个叫 bonusPlaneDelay 的设置。用来设定游戏多长时间出现一架奖励飞机。我们不能把奖励飞机的出现频率调得太高，要不然，就对敌机不公平了。奖励飞机出现的时间间隔是以帧为单位计算的。我们会用一个 bonusPlaneFrameCounter 变量去记录奖励飞机相隔多久才出现。



```
1. private function checkBonusPlane():void
2. {
3.     bonusPlaneFrameCounter++;
4.     if ( bonusPlaneFrameCounter >= bonusPlaneDelay)
5.     {
6.         if (bonusPlaneArray.length < maxBonusPlanes)
7.         {
```

现在，我们开始创建奖励飞机的实例。第一个要做的事情就是设置始点的 y 坐标。我们会用一个介于 0 到 bonusPlaneMaxY 的随机数来作为始点的 y 坐标。这样就能控制飞机最大的飞行高度。始点的 x 坐标是 -32,终点的横坐标是 gameWidth+32，终点的 y 坐标和始点的 y 坐标是一样的。

```
var randomY:int = Math.floor(Math.random()*bonusPlaneMaxY);
```

我们也需要设定奖励飞机的速度。越高的关卡，速度越高。当然，也不能太快，要不玩家就很难击毁奖励飞机了，所以这个速度是有上限的。剩下唯一要做的就是创建奖励飞机的实例和把它的引用放到 bonusPlaneArray 中。记得，当玩家击毁一架奖励飞机时候要给玩家加上 10 个炮弹，BonusPlane 类的构造函数中最后的一个参数就是这个值了。BonusPlane 类的构造函数可以用不同的参数给予不同数量的炮弹奖励，不过这里我们都设成 10。最后就是用 addChild()把奖励飞机添加到舞台上，并把引用放到 bonusPlaneArray 中，其实和创建敌机的过程都差不多。

```
8.         tempBonusPlane =new
BonusPlane(-32,randomY,gameWidth+32,randomY,bonusPlaneSpeed,10);
9.         bonusPlaneArray.push(tempBonusPlane)
10.        this.addChild(tempBonusPlane);
11.    }
12.    else
13.    {
14.        bonusPlaneFrameCounter = 0;
15.    }
16. }
17. }
```

## 更新对象

update()用来更新游戏中所有会移动的对象和创建炮弹的爆炸效果。这个函数有两部分的功能，第一个功能是先处理 Shot 类对象和他们各自的爆炸效果。第二部分是更新其他对象。



```
1. private function update():void  
2. {
```

一开始,我们用一个 for:each 语句去遍历 shotArray。在这个函数里,我们对其他药更新的对象,也是用这个方法。

```
3.     for each (tempShot in shotArray)  
4.     {
```

当所有会移动的对象到达终点或者完成动画的,要把他们的已完成属性设为 TRUE。这样就方便我们去判断是否需要从屏幕上移除这些对象。对于 Shot 的对象,不只是单单从屏幕上移除,还要创建 Flak 的实例来展现爆炸效果和表明终点。

由于 Flak 是 32\*32 的,它的注册点是(0,0),我们需要减去宽度和高度的一半,这样,爆炸的效果才会以炮弹的终点为中心。

```
5.         if (tempShot.finished)  
6.         {  
7.             tempFlak = new Flak();  
8.             tempFlak.x = tempShot.x-(tempShot.width/2);  
9.             tempFlak.y = tempShot.y-(tempShot.height/2);
```

现在,我们会把爆炸效果添加到舞台上,并把其引用放到 flakArray 中(看图【5-8】)。之后,我们就会抛出 CustomEventSound.PLAY\_SOUND 事件(其中以 Main.SOUND\_EXPLODE\_FLAK 作为参数)来播放声音。注意,我们要把音量设置为.5。

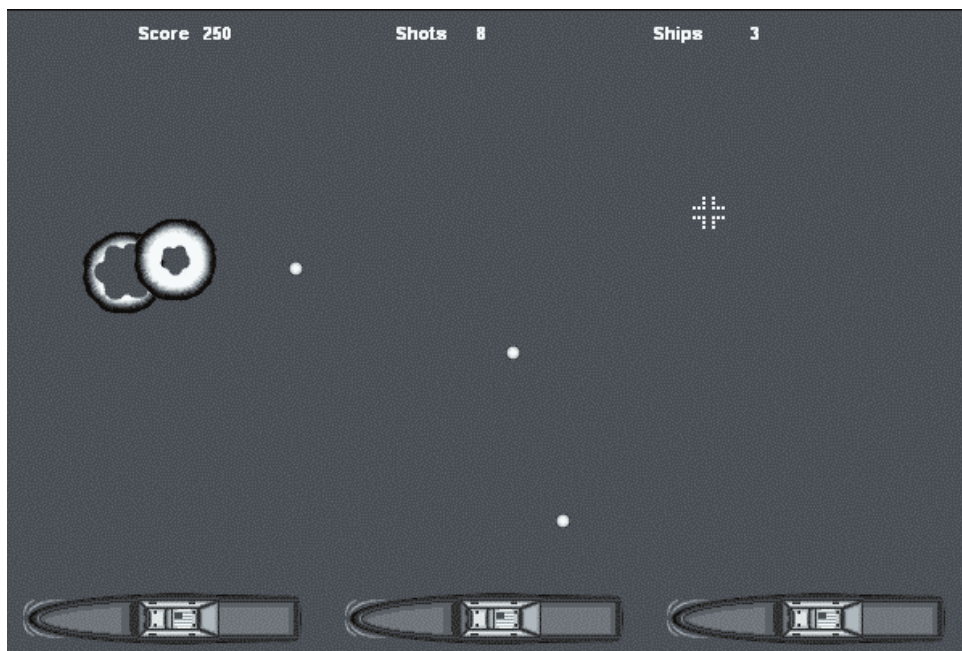
这是因为爆炸的声音会比其他的声音响亮一点。本来是应该做一个声音管理程序的,不过这里这样调整声音只是想告诉你,可以在代码中调整声音的相关属性。

然后,我们调用 removeItemFromArray(),给这个函数传递两个参数,第一个是要去掉的对象,第二个是这个对象所在的数组。我们会在稍后讨论这个很有用的函数。总之,这个函数会完成消除一个 Shot 实例的所有工作。

```
10.            this.addChild(tempFlak);  
11.            flakArray.push(tempFlak);  
12.            dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND,  
Main.SOUND_EXPLODE_FLAK,false,1,0,.5));  
13.            removeItemFromArray(tempShot,shotArray);  
14.        }  
15.    else  
16.    {  
17.        tempShot.update();  
18.    }  
19. }
```



如果炮弹还没有达到终点(`finished==false`), 就只是调用 `update()` 去更新对象。更新到的信息会在 `render()` (游戏循环的下一个函数) 中用到。 `update()` 会设置对象的下一个点的坐标。至于对象在屏幕上的移动, 我们会在下一节的 `render()` 函数中讨论。



图【5-8】

现在, 我们需要处理一下 `Flak`。和 `Shot` 类的处理不同, 当 `Flak` 的爆炸结束, 我们只是简单的把它从屏幕上去掉。我们不创建任何东西或者播放声音。你会看到, 剩下的对象我们都会进行单独的处理。

```
1.   for each (tempFlak in flakArray)
2.   {
3.       if (tempFlak.finished)
4.       {
5.           removeItemFromArray(tempFlak, flakArray);
6.       }
7.       else
8.       {
9.           tempFlak.update();
10.      }
11.  }
```

更新 `Enemy`, `Explosion`, 和 `BonusPlane` 对象

```
1.   for each (tempEnemy in enemyArray)
2.   {
3.       if (tempEnemy.finished)
4.       {
```



```
5.         removeItemFromArray(tempEnemy,enemyArray);
6.     }
7.     else
8.     {
9.         tempEnemy.update();
10.    }
11. }
12. for each (tempExplode in explodeArray)
13. {
14.     if (tempExplode.finished)
15.     {
16.         removeItemFromArray(tempExplode,explodeArray);
17.     }
18.     else
19.     {
20.         tempExplode.update();
21.     }
22. }
23. for each (tempBonusPlane in bonusPlaneArray)
24. {
25.     if (tempBonusPlane.finished)
26.     {
27.         removeItemFromArray(tempBonusPlane,bonusPlaneArray);
28.     }
29.     else
30.     {
31.         tempBonusPlane.update();
32.     }
33. }
34. }
```

## 去除对象

现在，我们来创建 `removeItemFromArray()` 函数。这是一个通用的函数，有两个参数：

- 1.item:从数组中要去掉的对象
- 2.group:第一个参数所在的数组

这个函数会利用 `Array.indexOf()` 来找到 item，然后调用 item 的 `dispose()`，再用 `removeChild()` 把它从屏幕上去掉。最后利用 `Array` 的 `splice()` 把它从数组中去掉。



这个函数一点也不完美。如果第一个参数的类型不是 Object，而是其他明确类型会更好。不过，这个函数为我们节省了很多时间和代码，所以用起来还是比较方便的。

```
1. private function removeItemFromArray(item:Object, group:Array) :void {  
2.   var index:int = group.indexOf(item);  
3.   group[index].dispose();  
4.   removeChild(group[index]);  
5.   group.splice(index, 1);  
6. }
```

## 检查位图碰撞

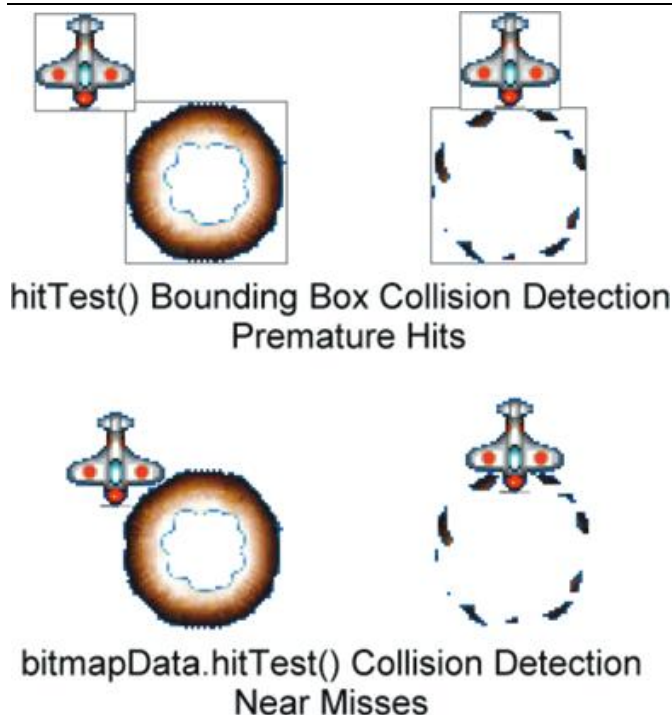
好的，现在来看看整个游戏中最重要的函数 checkCollisions()。这个函数里面需要的数据我们都准备好，是用来判断碰撞的。同时这个函数也非常复杂，所以我们要慢慢理解。

要完成这个碰撞检测，我们打算利用 Flash 的一个内建函数，对 ActionScript 开发者来说都比价陌生，同时，这个函数是属于 MovieClips 的。在 ActionScript 之前的版本（AS3 中也有），

你能够用 MovieClip.hitTest()去做碰撞检测。原理是利用一个 bounding box 碰撞检测。就是给图形做一个外接矩形，利用这个外接矩形做碰撞检测。这样，碰撞检测的效率就会比较高，因为形状规则。但是同时

也牺牲了碰撞的准确性。这个方法对于规则图形之间的碰撞检测非常有效，但是对于我们这个游戏，有复杂的不规则图形，用 bounding box 的话就不够准确了。甚至会影响到游戏的娱乐性。

看图【5-9】



图【5-9】

当 ActionScript 支持 Bitmap 的同时，也包含 BitmapData 的一个函数 hitTest()，除了名字和 MovieClip.hitTest()一样，没有什么是一样的了。

BitmapData.hitTest()是基于图像像素做检测的，而不是基于 bounding boxes。如果你的一张图片是有透明颜色的，在碰撞检测的时候，BitmapData.hitTest()

会忽略这个颜色。看图【5-9】。

因为我们已经给对象创建了各自的 BitmapData，所以我们可以用它们来做碰撞检测。使用 BitmapData.hitTest()有个缺点。当你用一个位图图片和另一个图片做像素级的碰撞检测，执行这个操作的位图信息只会创建一次，当

位图发生改变（例如旋转），这些信息是不会改变的。如果你真的要改变图片的一些信息，位图信息还是会保持原来的状态，这样就会发生不正确的碰撞。好消息是，我们可以手动更新位图信息。（在这本书的后面，我们会讨论这个问题）。

由于我们不需要旋转图片或者做其他的改变，所以我们不用担心这个问题。我们直接用 BitmapData.hitTest()就好了。

我们把目光放到 checkCollisions()中。当你需要做多种对象的碰撞测试，把这些对象当做一个群体来处理会是个不错的方法，如果可以的话，可以同时做碰撞检测。你可以在遍历这些对象的时候陷入一个循环来做。

如果这些循环嵌入正确，那每次循环都能做最多的碰撞检测并把循环次数控制到最少。然而，这也是多个嵌入循环最难控制的一点。我们在这个游戏里使用一个保守的方法。我们已经创建了 3 个单独的循环，



每一个循环都说明一组碰撞。

优化这个函数有很多方法，但我们希望你能够看明白代码，而不是只看到一些复杂的嵌套循环。我们现在来试试在这个函数中测试 3 个精确位图的碰撞，有下面的情况：

1. 敌机和炮弹相撞，奖励飞机和炮弹相撞

2. 敌机和船队碰撞。

一开始，我们先声明几个变量去获取我们要测试的数组对象的长度。

```
1. private function checkCollisions():void
2. {
3.     var enemyLength:int = enemyArray.length-1;
4.     var flakLength:int = flakArray.length-1;
5.     var shipLength:int = shipArray.length-1;
6.     var bonusPlaneLength:int = bonusPlaneArray.length-1;
```

接下来，我们从后面开始遍历 enemyArray 数组。调用之前创建的 tempEnemy 变量，这样，每执行一次循环体，我们就不需要浪费时间和资源去创建一个新的变量。注意一下 for 循环前面的 enemy:，这是一个标志。

当做碰撞测试的时候，如果数组中的这个项已经不再数组中，我们就会用这个标志去跳出循环（当讲到 break 语句的时候会解释一下）。

```
7.     enemy: for (var ctr2:int = enemyLength; ctr2 >= 0; ctr2--)
8.     {
```

我们第一个要做测试的就是炮弹爆炸和敌机。首先，我们需要创建一个 Point 类对象。每一个要做碰撞测试的对象都需要一个 Point 类。我们把这个 Point 命名为 enemyPoint。enemyPoint 就是敌机对象的左上角的点。

bitmapData.hitTest()需要这个点的坐标去做它的一个参数。接着，我们开始遍历 flakArray。

```
9.         tempEnemy = enemyArray[ctr2];
10.        var enemyPoint:Point = new Point(tempEnemy.x, tempEnemy.y);
11.        for (var ctr:int = flakLength; ctr >= 0; ctr--)
12.        {
13.            tempFlak = flakArray[ctr];
```

我们也需要第二个 Point 用来表示炮弹爆炸效果图的左上角。这个变量也是作为 bitmapData.hitTest() 的一个参数，用来确定像素级的碰撞检测从哪里开始。

```
14.            var flakPoint:Point = new Point(tempFlak.x, tempFlak.y);
```

下面是这个函数的最主要的部分。碰撞检测要来了（这部分包含在一个 if 语句中）：



```
15.         if
            (tempFlak.image.bitmapData.hitTest(flakPoint,255,tempEnemy.image.bitmapData,enemy
            Point))
16.         {
```

现在，停一停，让我们先来了解一下这里究竟发生了什么事情

flakPoint:做碰撞检测的炮弹爆炸的左上角。

255:透明系数（从 0 到 255，此处是不透明）

tempEnemy.image.bitmapData：我们要测试碰撞的敌机

ePoint：我们要测试碰撞的敌机的左上角。

如果发生碰撞，我们就会执行下面的代码。第一件事就是要调用之前创建的 `removeItemFromArray()`，并把 `tempEnemy` 和 `enemyArray` 传进去。注意，我们没有把炮弹爆炸的效果移除，这是因为只要这个效果还在屏幕上，它就有可能击毁其他的敌机。然后，我们抛出一个 `CustomEventSound` 事件，来播放 `Main.SOUND_EXPLODE_PLANE` 这个音效。

```
17.         removeItemFromArray(tempEnemy,enemyArray);
18.         dispatchEvent( new
            CustomEventSound(CustomEventSound.PLAY_SOUND,
            Main.SOUND_EXPLODE_PLANE,false,1,0,.5));
```

下面这两行代码涉及到的函数，我们会在这一节的后面讲到。`makeExplosions()`会在指定的位置（由传入的两个参数决定）创建一个炮弹爆炸的实例。

然后，调用 `addToScore()`函数把相应的分数加给玩家。除了固定的 100 分（摧毁一个敌机的得分）之后，如果同一次炮弹爆炸中有摧毁其他的敌机，就多奖励 500 分。举个例子，一个炮弹摧毁了 2 个敌机，会得到 700 分（ $100*2+500$ ），摧毁 3 个敌机，会得到 800 分（ $100*3+500$ ）。

```
19.         makeExplosion(tempEnemy.x,tempEnemy.y);
20.         addToScore(baseEnemyScore+(tempFlak.hits*enemyHitBonus));
```

接着，我们就更新当前的 `Flak` 对象的 `hits` 属性，这样，当玩家一次摧毁不止一个敌机的时候，就可以给玩家添加奖励。我们把 `hits` 设置为 `Flak` 的成员属性，而不是一个局部变量，这是因为 `Flak` 的爆炸效果会持续多帧，会在这个碰撞检测函数中一直存在。

```
21.         tempFlak.hits++;
```

最后，我们会利用我们上面提到的标签。`break` 语句会让我们跳出当前循环，跳到 `enemy` 标签。这样，这个敌机就不用和其他的炮弹做碰撞测试了。然后，我们又从一个新的敌机开始，和其他的炮弹做碰撞检测。这样做，Flash 运行时就不会因为引用一个不存在的空对象而报错。

```
22.         break enemy;
23.     }
```



```
24.     }  
25. }
```

现在，我们开始奖励飞机和炮弹之间的碰撞检测了。这里也需要做一次循环。然而，由于屏幕上不是总有奖励飞机，所以不需要在每一帧上进行这个检测。

奖励飞机的 `bitmapData.hitTest()` 几乎和敌机的 `bitmapData.hitTest()` 一样。不过，当击中奖励飞机的时候，我们会播放另一个声音(`Main.SOUND_BONUS`)，并把 `tempBonusPlane.bonusvalue` 的值添加到玩家的弹药量中。之后就抛出一个事件，告诉计分板弹药量要更新了。还有一个不同就是，给玩家加分的时候是加 `bonusPlaneScore` 而不是 `baseEnemyScore`。

注意：我们希望爆炸的效果会覆盖在奖励飞机上面，但是由于飞机的大小是 `64*64`，爆炸的大小是 `32*32`，我们需要让爆炸的效果移动宽度的一般和高度的四分之一。

```
1.     bonusplane: for (ctr2 = bonusPlaneLength; ctr2 >= 0; ctr2--)  
2.     {  
3.         tempBonusPlane = bonusPlaneArray[ctr2];  
4.         var bonusPlanePoint:Point = new Point(tempBonusPlane.x, tempBonusPlane.y);  
5.         for (ctr = flakLength; ctr >= 0; ctr--)  
6.         {  
7.             tempFlak = flakArray[ctr];  
8.             flakPoint = new Point(tempFlak.x, tempFlak.y);  
9.             if (tempFlak.image.bitmapData.hitTest(flakPoint,255,  
tempBonusPlane.image.bitmapData,bonusPlanePoint))  
10.            {  
11.                dispatchEvent( new  
CustomEventSound(CustomEventSound.PLAY_SOUND,  
Main.SOUND_BONUS,false,1,0,1));  
12.                shots += tempBonusPlane.bonusValue;  
13.                dispatchEvent(new  
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC  
ORE_BOARD_SHOTS,String(shots)));  
14.                makeExplosion(tempBonusPlane.x +  
tempBonusPlane.width/2,tempBonusPlane.y + tempBonusPlane.height/2);  
15.                addToScore(baseBonusPlaneScore+(tempFlak.hits*enemyHitBonus));  
16.                tempFlak.hits++;  
17.                removeItemFromArray(tempBonusPlane,bonusPlaneArray);  
18.                break bonusplane;  
19.            }  
20.        }  
21.    }
```

其实，我们可以把奖励飞机看做是一个特别的“敌机”，这样，我们就不需要创建一个新的类了，而且，



在做上面的碰撞检测的时候，我们只需要做一次循环就好了。

尝试去优化吧，这样对你的游戏开发技术很有好处。

现在，我们需要去做敌机和玩家船队的碰撞检测了。唉，我们又要再次遍历 enemyArray 数组了，不过幸运的是，由于玩家的船只总是坐落在一个固定的 y 坐标。

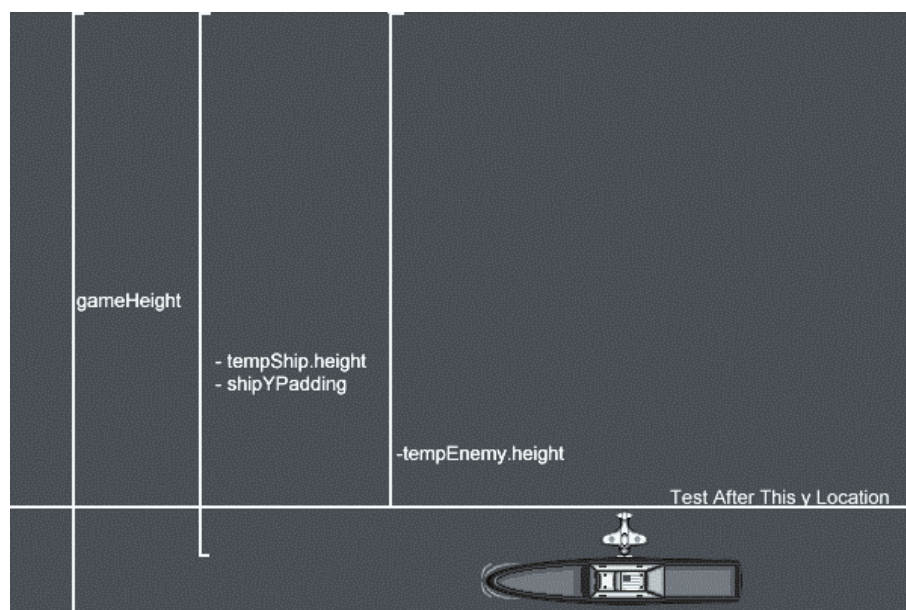
这样，我们就可以根据敌机当前的 y 坐标来判断时候需要做碰撞检测了。

注意在 for 循环之前的 ship。这是另外一个标签。当我们正在检测的的对象从数组中被移除，我们就会利用这个标签跳出循环。下面的大部分代码对于你来说应该也是比较熟悉了。

```
22.     ship: for (ctr = shipLength; ctr >= 0; ctr--)
23.     {
24.         tempShip = shipArray[ctr];
25.         var sPoint:Point = new Point(tempShip.x, tempShip.y);
26.         enemyLength = enemyArray.length-1;
27.         for (ctr2 = enemyLength; ctr2 >= 0; ctr2--)
28.         {
29.             tempEnemy = enemyArray[ctr2];
```

这里就是这个循环有趣的地方了。我们会检测敌机当前的 y 坐标与船只的距离是否能够发生碰撞。因为船只总是在一个固定的地方，所以我们要做的事情就是计算出敌机可以碰到船只的临界点。我们会用 gameHeight 减去船只的高度，再减去 shipYPadding，最后再减去敌机的高度。图【5-10】会展示这样计算的理由。

```
30.         if (tempEnemy.y >
31.             gameHeight-tempShip.height-tempEnemy.height-shipYPadding)
32.         {
33.             enemyPoint = new Point(tempEnemy.x, tempEnemy.y);
34.             if
35.             (tempShip.image.bitmapData.hitTest(sPoint,255,tempEnemy.image.bitmapData,enemyP
36.             oint))
37.             {
```



图【5-10】

在这里检测碰撞，和前面的有点不同。首先，我们会移除敌机。和 Flak(炮弹)不同，敌机只是要摧毁船只。

然后，我们抛出 CustomEventSound.PLAY\_SOUND 事件去播放 Main.SOUND\_EXPLODE\_SHIP 这个声音。接着创建 3 个爆炸的效果。只是用来表明船只被摧毁了。

不过这还不够，我们要调用 removeItemFromArray() 移除被摧毁的船只，抛出一个事件告诉计分板更新关于船只数量的信息。最后，可以跳出 ship 这个循环。

```

35.         removeItemFromArray(tempEnemy,enemyArray);
36.         dispatchEvent( new
CustomEventSound(CustomEventSound.PLAY_SOUND,Main.SOUND_EXPLODE_SHIP,fals
e,1,0,1));
37.         makeExplosion(tempEnemy.x,tempEnemy.y);
38.         makeExplosion(tempShip.x+25,tempShip.y+5);
39.         makeExplosion(tempShip.x+tempShip.width-40,tempShip.y+10);
40.         makeExplosion(tempShip.x+tempShip.width/2,tempShip.y+3);
41.         removeItemFromArray(tempShip,shipArray);
42.         ships--;
43.         dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC
ORE_BOARD_SHIPS,String(ships)));
44.         break ship;
45.     }
46. }
```



```
47.     }  
48.     }  
49. }
```

我们已经写好了 checkCollisions()函数，不过还需要创建一些函数去支持我们刚刚添加的代码。第一个函数是 createExplosions()。这个函数需要一个坐标值 ( x , y ) 作为参数，并在(x,y)坐标上放置一个 Explosion 的实例（爆炸效果）。这个操作非常简单明白。我们在之前也已经用过类似的代码了。这里就不详细说了。

```
1. private function makeExplosion(explodeX:int,explodeY:int):void  
2. {  
3.     tempExplode = new Explosion();  
4.     tempExplode.x = explodeX;  
5.     tempExplode.y = explodeY;  
6.     this.addChild(tempExplode);  
7.     explodeArray.push(tempExplode);  
8. }
```

addToScore()会在玩家摧毁敌机获得分数的时候被调用。addToScore()有一个参数，就是要添加的分数，抛出一个事件，告诉计分板更新玩家的分数。也会调用 checkBonusShips()（在后面会说到）。

去判断玩家是否获得足够的分数去获得一只额外的船只。

```
1. private function addToScore(val:Number):void  
2. {  
3.     score += int(val);  
4.     checkBonusShips();  
5.     dispatchEvent(new  
        CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC  
        ORE_BOARD_SCORE,String(score)));  
6. }
```

checkBonusShips()只是做一个简单的计算，判断玩家是否已经到达奖励一个船只所需的分数。不过，这里有个小技巧看起来不是那么好理解。

我们之前已经创建了一个难度设定 scoreNeededForExtraShip。在这个游戏中，scoreNeededForExtraShip 的值是 10000（不过你可以自己设置的）。

看起来，我们只需要判断玩家的分数是否是 scoreNeededForExtraShip 的某个倍数就可以了（例如 10000,20000,30000），如果是，就奖励一个船只。

但是，有可能出现这样一种情况，玩家的分数比 scoreNeededForExtraShip 的某倍小或者大，那么玩家就没有获得奖励的船只了。要不这样，如果我们



看到玩家的分数大于或者等于 scoreNeededForExtraShip 的某个倍数，我们就给玩家奖励一个船只。

```
1. private function checkBonusShips():void
2. {
3.     if ((score-(extraShipCount*scoreNeededForExtraShip) >= scoreNeededForExtraShip))
4.     {
5.         ships++;
6.         extraShipCount++;
7.         dispatchEvent( new
CustomEventSound(CustomEventSound.PLAY_SOUND,Main.SOUND_BONUS_SHIP,false,
1,0,1));
8.         dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SC
ORE_BOARD_SHIPS,String(ships)));
9.     }
10. }
```

## 渲染对象

render()由 runGame()调用，用来把对象渲染到屏幕上。对于会移动的对象(Shot, Enemy, 和 BonusPlane)和动画 ( Flak 和 Explosion )，各自都用一个 for:each 循环。在这里可以用 for:each，因为我们不用从数组上移除任何对象，我们只是调用它们各自的渲染函数。

```
1. private function render():void
2. {
3.     for each (tempShot in shotArray)
4.     {
5.         tempShot.render();
6.     }
7.     for each (tempFlak in flakArray)
8.     {
9.         tempFlak.render();
10.    }
11.    for each (tempEnemy in enemyArray)
12.    {
13.        tempEnemy.render();
14.    }
15.    for each (tempExplode in explodeArray)
16.    {
```



```
17.         tempExplode.render();
18.     }
19.     for each (tempBonusPlane in bonusPlaneArray)
20.     {
21.         tempBonusPlane.render();
22.     }
23. }
```

## 过关了

在 `runGame()` 中最后的一个函数就是 `checkForEndOfLevel()`。这个函数判断玩家是否满足过关条件。有两个变量决定玩家是否已经过关了：

`enemyArray` 的长度和 `incomingCount`。`incomingCount` 用来判断创建的敌机数量是否等于难度设定 `numEnemies`。如果是的话，就表明在这关里面已经完成创建所有的敌机了。不过，还可能会有敌机在屏幕上的。由于每个在屏幕上的敌机都会在 `enemyArray` 中有记录，所以我们只需要判断 `enemyArray` 的长度就知道屏幕上还有没有敌机了。

还没完呢。即使所有的敌机都不在了，突然结束的话对玩家来说就不够和谐了。如果屏幕上还有爆炸和炮弹和奖励飞机在飞行，我们就需要等待这些对象（的动画）结束了才能开始新的一关。这同时也给了玩家一个机会在最后的时刻去射击奖励飞机。我们要检查对象的各个关联数组的长度。还有，每一关结束的时候，要给玩家加上剩余炮弹的数量乘以十的分数。最后，我们抛出 `Game.NEW_LEVEL` 事件让 `Main` 去调用 `newLevel()`。

```
1. private function checkforEndLevel():void
2. {
3.     if (enemyArray.length <= 0 && incomingCount >= numEnemies &&
        explodeArray.length <=0 && flakArray.length <=0 && shotArray.length <=0 &&
        bonusPlaneArray.length <= 0)
4.     {
5.         addToScore(10*shots);
6.         cleanUpLevel();
7.         dispatchEvent(new Event(Game.NEW_LEVEL));
8.     }
9. }
```

你会注意到在这些处理函数里面有一个函数叫 `cleanUpLevel()`。这个函数清除屏幕上所有的东西和监听玩家射击的侦听器。我们把这些分离到第二个函数 `checkForEndGame()` (后面会提到)。

这个函数里面的代码对于你来说应该比较熟悉了。和 `render()` 非常相似，需要遍历所有对象相关的数组。不过，不同的是，我们不能够用 `for:each` 了。

在函数的最后，要移除 `MouseEvent.MOUSE_DOWN` 的侦听器。



```
1. private function cleanUpLevel():void
2. {
3.     var ctr:int = 0;
4.     for (ctr = shotArray.length-1;ctr >=0;ctr--)
5.     {
6.         removeItemFromArray(shotArray[ctr],shotArray);
7.     }
8.     for (ctr = flakArray.length-1;ctr >=0;ctr--)
9.     {
10.        removeItemFromArray(flakArray[ctr],flakArray);
11.    }
12.    for (ctr = enemyArray.length-1;ctr >=0;ctr--)
13.    {
14.        removeItemFromArray(enemyArray[ctr],enemyArray);
15.    }
16.    for (ctr = explodeArray.length-1;ctr >=0;ctr--)
17.    {
18.        removeItemFromArray(explodeArray[ctr],explodeArray);
19.    }
20.    for (ctr = shipArray.length-1;ctr >=0;ctr--)
21.    {
22.        removeItemFromArray(shipArray[ctr],shipArray);
23.    }
24.    for (ctr = bonusPlaneArray.length-1;ctr >=0;ctr--)
25.    {
26.        removeItemFromArray(bonusPlaneArray[ctr],bonusPlaneArray);
27.    }
28.    stage.removeEventListener(MouseEvent.MOUSE_DOWN, shootListener);
29. }
```

runGame()最后调用的函数是 checkForEndGame()。这个函数很简单。它判断 ships 是否小于等于 0。为什么是小于等于 0 而不是等于 0 呢？

这是因为万一出了什么问题，玩家的船只数量小于 0 了，游戏就不用结束了，小于等于 0 是为了避免这种情况发生。剩下的代码都很容易明白。首先，抛出一个 Game.GAME\_OVER 事件去告诉 Main.as 可以开始结束游戏。然后用 removeChild(crosshairs)把识字瞄准器从屏幕上移除，并用 Mouse.show()把鼠标再次展示出来，移除 MOUSE\_MOVE 的侦听器。接着调用 cleanUpLevel()函数去清除屏幕上的显示对象，最后把 crosshairs 设为 null，这样在游戏重新开始的时候就会被再次创建。

```
1. private function checkforEndGame():void
2. {
3.     if (ships <=0)
```



```
4.      {
5.          dispatchEvent(new Event(Game.GAME_OVER));
6.          removeChild(crosshairs);
7.          Mouse.show();
8.          stage.removeEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
9.          cleanUpLevel();
10.         crosshairs = null;
11.     }
12. }
```

经过第二章和第三张关于游戏框架的讲解，还有这个游戏的具体实现过程，相信你对一个游戏是怎样制作会有有一个大概的印象吧。

好了，你可以开始玩游戏了。在玩的时候，注意一下你喜欢和不喜欢的地方。

## 总结

在上面这两章，我们花费了很多的时间去讲解如何在游戏中运用之前说到的游戏框架。而且也模仿了 Missile Command，创建了我们的游戏。

在游戏中，有很多地方你可以修改一下，让游戏变得更加好玩。

如果你闲来无事，想改一下游戏，那么调整游戏难度应该是你最好的开始。或者你可能会调整敌机的飞行速度，玩家一开始的弹药量，获得额外船只需要的分数等等，注意当你作出这些调整的时候对游戏的影响。下面是我列出来的一些地方：

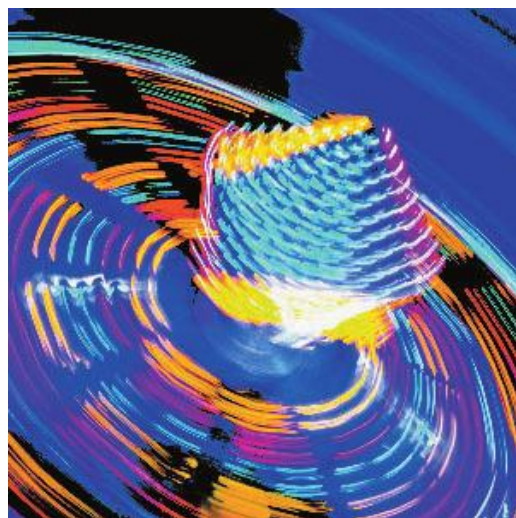
更多的难度设置：你还可以添加其他的难度设置，例如奖励飞机的分值，每一关结束后根据剩余的弹药量来加分。

保存船只：当三只船都被击毁，而玩家还有额外的船只的时候，游戏会继续进行，直到下一关开始的时候才会把船只放出来。或许你可以使用别的机制。

船只开炮：你可以让船只拥有开火的能力，这样就可以不用“隐形”的大炮了。

添加其他不同的敌人：游戏中就只有一种敌机，或者你可以添加其他类型的敌机，例如要 2 炮才能击毁的。

在玩游戏的过程中，你或者会留意这个游戏一些做得好和做的不好的地方。有很多地方可以进行优化，优化的过程会很大的提高你自己的编程能力的。



## 第六章:

# 预备！坦克大战！

---

在前两章中,我们探究了怎么用一些高级技术来完成一个相对简单的游戏.在这一章和下一章中,我们将进一步增加难度并且在更高的一个层次上开发并优化我们的 flash 游戏.

在这一过程中,我们会接触到很多东西,我们会很快的介绍并简单的解释一下这些高级话题.首先,我们将要探索一下怎么样创建并使用一个 tile sheet,并且尽量编写的整洁.在本章中,我们将学习 tile sheet 并学习怎么样通过该技术实现游戏世界的地图;接着我们会学习怎么样使用一些免费和开源的工具来创建不同的游戏地图.然后我们会接触到怎么样通过 copyPixels 和 blitting 方法来优化 tile.

在下一章我们讨论怎么在逻辑上让玩家和敌人显示在基于 tile 实现的游戏地图上,一些人工智能,让玩家和敌人移动和射击.这一过程会添加一些新的类和重用框架,并且通过创建一些新的 Main.as 来整合游戏和框架.



## 设计坦克大战

我们简要做的一个典型的迷宫,追捕风格的游戏.看起来就像 Namco 公司的吃豆游戏.在我们的游戏中,玩家就是一个坦克.其任务是找到那件被盗的宝物并消灭所有的敌军坦克并且保证不要被消灭.

迷宫将由一些可移动和不可移动 tile 构成.当然还要有子弹,多条命.我们将要创建的游戏引擎会支持任意数量(等级)的关卡.只要关卡设计者遵从一些简单规则,就可以在游戏引擎中良好运行.

在很多这类游戏引擎中,开发者们不厌其烦的用 tile 来表示数据就是为了给预编码更多的灵活性.我们将会去探索怎么样用基于 tile 的技术创建一个名为 坦克大战的游戏.例如,一个可支持 4 方向运动的路径,当坦克移动到这个十字路口时,它就要告诉游戏引擎坦克所有可以移动的方向.用这种方法来编写迷宫基础运动的好处就是简洁.迷宫游戏是硬编码(就是指直接做为固定内容写入了程序)而且我们也没有必要去开发一个在任何条件下都通用的东西.缺点就是一旦开发者创建并且将移动逻辑硬编码到迷宫中后如果再添加新等级的迷宫地图还要继续重复第一次的工作.

我们并不采用这种方法.我们会创建一个游戏关卡引擎并且设置一个移动算法来动态的完成这些工作--这就意味着一旦我们为迷宫移动创建了逻辑规则,所有的游戏等级都将遵从这些逻辑规则.我们会创建一个灵活,适应性高的关卡引擎.

## 下面是一些游戏的基础信息:

**游戏名称:** 坦克大战

**游戏类型:** 迷宫/追捕/射击

**游戏简介:** 玩家在一个 2D 迷宫中找寻被偷的宝物.敌军坦克会在迷宫中驻守包围那个宝物.

**玩家目标:** 收集到每一关的宝物才能继续下一关.玩家的弹药量是有限的,但是可以从那些粗心的敌人那里收集到他们遗失的弹药.很意外吧?玩家的坦克和敌军的坦克使用的是同一型号的弹药.玩家开始的时候有 3 辆坦克(命),伴随着玩家通关数的增加玩家会得到更多的坦克.

**敌军描述:** 一旦玩家进入敌军坦克的警戒区他们就会追捕并对玩家开火.游戏区域将会被分为 4 个区.敌军坦克配备着一个简单的迷宫追捕 AI 并且会不停的巡视来寻找玩家.

**敌军目标:** 摧毁玩家坦克.

**过关条件:** 当玩家成功收集到宝物时本关结束.



**游戏结束条件:**当玩家所有的坦克(命)都被消灭时.

**难度:**游戏难度和等级相关.伴随着等级的提升,游戏关卡设计师可以适当的添加更多的坦克到关卡中并且设置对应的弹药供应量.当然坦克的智能和设计速度也可以在游戏等级数据中设置.

## 第 6,7 章的游戏设计观:

下面是接下来 2 章中我们涉及到的一些简洁的游戏开发纲要:

### 第六章:

使用 title sheet 来处理游戏图像.

通过图形编程来创建一个 tile sheet

使用 Mappy Level 编辑器来创建关卡.

创建一个 flex 项目类库.

创建 TileSheet 类

创建基础的 Level 类

扩展 Level 类来支持不同的数据

将游戏中的所有 tile 融合到一个 BitmapData 中然后绘制输出到屏幕上.

### 第七章:

为游戏对象创建 BlitSprite 类

根据迷宫游戏的基础逻辑规则扩展 BlitSprite 类

使用 bitmap tile 来驱动 Sprite

在游戏关卡中设置玩家和敌人的位置.

根据区域(zone)和追捕逻辑来创建基础的追捕 AI



设置弹道为直线

使用 look-ahead 变量来做碰撞检测和移动.

完成游戏声音和显示.

为 No Tanks 设置框架

## 在项目中添加游戏素材文件夹(assets)

在 Flak Cannon 中,我们使用的素材是从 Spritelib GPL 中获取.我们还会继续它们,但是这次我们将把所有素材放到一个 32x32 的 sheet 中.我们仍然使用 SFXR 工具来为我们的游戏创建音效.

## 使用 Spritelib GPL

就像上一个游戏一样,我们仍然使用由 Ari Feldman 维护的 Spritelib GPL( <http://www.flyingyogi.com/fun/spritelib.html>.) 这是一个无版权图像库,这就意味你可以任意的使用其中的图像(将他们存储到另一个库中).本书中的好几个游戏都是使用了这个库.

## 使用 tile sheet

对于 as 游戏开发者,其中一个最有效地技巧就是使用原生位图来渲染屏幕图像.一个 tile sheet 就是一个由位图组成的文件(GIF,JPG,PNG,等等),这些图像被有序的组织成网格格式排布的格式.一般的 tile sheet 格式是由行列相等的正方形图像组构成.例如,我们将要用的图像是宽为 32 像素,高位 32 像素.我们将把这一类型的图片填充到一个 sheet 中.这些 tile 在我们的游戏中将会被用来创建背景和 sprite 类型动画.

相对于逐个将 32x32 的位图 tile 导入到 flash 库而言,我们只需要导入一个 tile sheet.然后我们只需要编写代码来获取我们所需要的那部分就可以了.这一部分将用 as3 中的 BitmapData 来完成.

tile sheet 将会被定义成一个包含了多个 tile 且固定尺寸的数组.如图 6-1 sheet 中左上角角落的那个就是 tile 0 了.

## 用 tile 来构造游戏关卡

我们的游戏关卡由一个 20x15 的网格组成,每个网格是大小为 32x32 像素的 tile.这个网格将被编写成一个由 tile ID 构成的二维数组.tile ID 的范围是 0-31(如图 6-1),这个数字代表了 tile 在 sheet 上的 id.二维数组我们可以将其理解为包了一个一维数组的数组.ActionScript 并没有内建的多维数组.我们通过把数组的每个元素替代成一个数组来构建二维数组.关卡的尺寸为 640(20 x 32) x 480 (15 x 32).这就意味着我



们的关卡数据将有 300(20 x15)个 tile ID 数字组成.我们将会有 15 行 20 列.下面就是一个我们将要创建的 2 维关卡数据:

```
var backGroundMap:Array = [
[26,24,25,0,26,26,26,26,26,0,0,26,26,26,26,26,0,24,25,26],
[27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,27],
[29,0,28,0,0,0,24,25,0,24,25,0,24,25,0,0,0,27,0,29],
[27,0,28,0,27,0,0,0,0,0,0,0,0,0,0,27,0,29,0,27],
[29,0,0,0,29,0,0,30,0,0,0,0,31,0,0,29,0,0,0,29],
[0,0,28,0,26,0,0,31,0,0,0,0,31,0,0,26,0,26,0,0],
[0,0,0,0,0,0,31,0,0,0,0,31,0,26,0,0,0,0,0],
[26,26,26,0,0,26,0,30,30,30,30,30,30,0,0,0,26,26,26],
[0,0,0,26,0,0,0,0,0,0,28,0,0,0,0,26,0,0,0],
[0,0,0,0,26,0,0,27,0,27,0,27,0,28,0,26,0,0,0,0],
[27,0,0,0,27,0,0,29,0,29,0,29,0,28,0,27,0,0,0,27],
[29,0,0,0,29,0,0,0,0,0,0,0,0,0,0,29,0,0,0,29],
[27,0,0,0,0,26,0,28,0,28,0,28,28,0,26,0,0,0,0,27],
[29,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,29],
[26,24,25,0,0,26,26,26,26,0,0,26,26,26,26,0,0,24,25,26]
```

[多维数组的解释跳过]

我们的数据将被组织为行列的形式.我们将会使用这个二维数组来存储关卡的数据.因为我们可以很容易的通过编程来获取 sheet 网格中指定 ID 的 tile,所以剩下的问题就是我们是否可以通过行和列检索到指定的 ID.行就是二维数组中第一个数组的下标,列就是第二个下标.例如:访问第 10 行第 5 列的方法是:

```
levelData[9][4] = 26
```

由于我们的数组是从 0 开始的,这就意味着第一个数据的下标从 0 开始,9 就是第 10 行,4 就是第 5 列.

如果你看的比较仔细,你可能会想起一些以前的事情.在我们的网格中一般用"行"来代表 y 轴,"列"来代表 x 轴,但是我们认为反过来更好.很多开发者为这个问题争论了很多年.就我们的经验看来,在 as3 中,用 [行][列]表示的二维数组更好理解.你只需要将数据倒过来就可以在关卡数组中查找到正确的 tile 了.

## 创建关卡

创建一个关卡就像拿起 tile 的 id 然后填入到二维的网格中一样简单.这一过程即使手动来做也非常简单,但是用地图编辑器会更简单.一些开发者发现将 tile 在绘制工具中或者甚至是在 flash ide 中来创建关卡会更为方便.其他人可能是通过基于 Flash 的关卡编辑器来创建.就我们的经验看来,我们会非常喜欢使用 Mappy 来创建关卡的(关于下载和使用一会儿我们会给出详细介绍).不幸的是,没有 Mac 版本 Mappy.他可以在 Parallels 或者 VMWare 的软件上面运行,但是没有原生的 Mac 版本.他同样可以在 Linex 和 Mac 系统上的

WINE 上面运行.即使你没有 Windows 的机器,你同样的可以使用 Mappy.当然也有很多其他的关卡编辑器可以去尝试,例如 FLAN.

## 使用 Mappy 创建关卡

首先我们在 Mappy 中将每种 tile(图片)组合成一个大图,然后导出 PNG 文件.接下来,我们会在 Mappy 中读取这些 tile,然后使用 Mappy 为我们的游戏创建一个关卡.之后我们会从 Mappy 导出这个关卡的数据.我们将会用这些数据构建显示我们的游戏关卡.

我们将使用下面的图像来作为我们游戏的素材.它们已经在 Spritelib GPL 被切割好,并且放到了我们的 tile sheet 中.如图 6-1

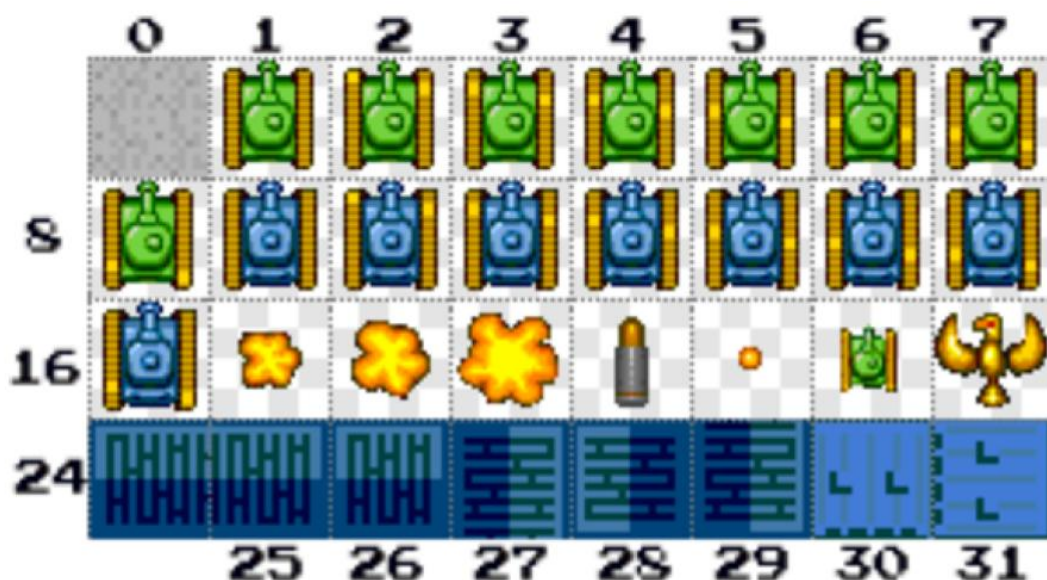


图 6-1 游戏中所有的 tile 素材(包含对应的 id 编号)

首先,我们将这个 tile sheet 命名为 tanks\_sheet.png. 它将被放到 我们 flex 项目的/assets 文件夹中. 对于 FLash IDE,它将被内嵌到我们的 fla 库中.

他是一个 256x128 背景透明的 PNG 文件.

让我们来逐行浏览并描述其中每一个 tile:

Row1(tiles 0-7):Tile 0 是一个代表路或者地板的 tile.他被用来表示可行走的背景;这就意味着玩家和敌人坦克都可以在上面行走.这一行仍然包含了坦克行走动作的前 7 帧.在每一帧中,坦克的履带都有轻微不同.这是因为当我们连续播放这些图片时想让坦克看起来是在行驶而非在屏幕上滑动.

Row2(tiles 8-15):这一行包含了玩家坦克的 8 帧图像和敌军坦克前 7 帧的图像.在每一帧中坦克的履带有略微的不同.就像刚刚玩家的坦克一样,是为了让坦克看起来像是在行驶而非在屏幕上滑动.



### Row3 (tile 16-23)

tile 16 是敌军坦克的最后一帧图像

tile 17-19 是爆炸的图像

tile 20 代表可拾取的弹药

tile 21 代表玩家或者敌军攻击对方时的子弹.

tile 22 代表可拾取的额外坦克(生命)

tile 23 代表玩家在每个关卡中都要去寻找的宝物.

### Row4(tiles 24- 31)

这一行包含了可以用来当做围墙的 8 中图像.玩家和敌军都不可以在上面移动.

## Sptites vs. tiles vs. 背景

我们将使用一组"sprite"来表示我们游戏中的任何交互对象.这些 sprite 不属于游戏关卡背景.他们会被放到前景里.sprite 由我们刚刚在 tile sheet 中看到的 tile 组成.游戏中背景将由可行走的路和墙壁组成.这些 sprite 将由坦克,可拾取的弹药,子弹,爆炸和游戏中其他的元素组成.

## 创建 tilesheet

一个 tilesheet 和一般的图像看起来是一样的,不过它是用一种独特方式建立的.我们的 tile sheet 将由一个宽 256 像素,高 128 像素的图像组成.这就意味着我们在这里面存放 4 行每行 8 个大小为 32x32 的 tile.为了便于观看,我们将直接从 Spritelib GPL 拷贝文件然后将他们放到我们 sheet 中.对于这个例子,每个人都会有他们想用的工具.在这里我们将要使用一款名为 GIMP 的工具来创建 tilesheet.PS 和 FW 也同样是可以达到目的的专业工具.

GIMP 是一个免费,社区支持且和 PS 或 FW 有同样功能的工具.你可以从

<http://www.gimp.org> 下载到最新版本.

其他更详细介绍已经超出了本书的范围,但是只要按照下面的步骤就足可以完成 创建和存储游戏 tile sheet 的工作.

你可以用任何工具来创建背景透明的 PNG 图像.PS 和 FW 都可以.其他的一个免费的工具也可以完成这项工作例如: Paint.net 和 Pixer.我们会介绍怎么用 GIMP 来完成,但是你没必要拘束于这个工具选择一个顺手的工具吧.

1.开启 GIMP 并且选择创建新图像.如图 6-2 勾选高级选项,注意设置背景填充为透明.

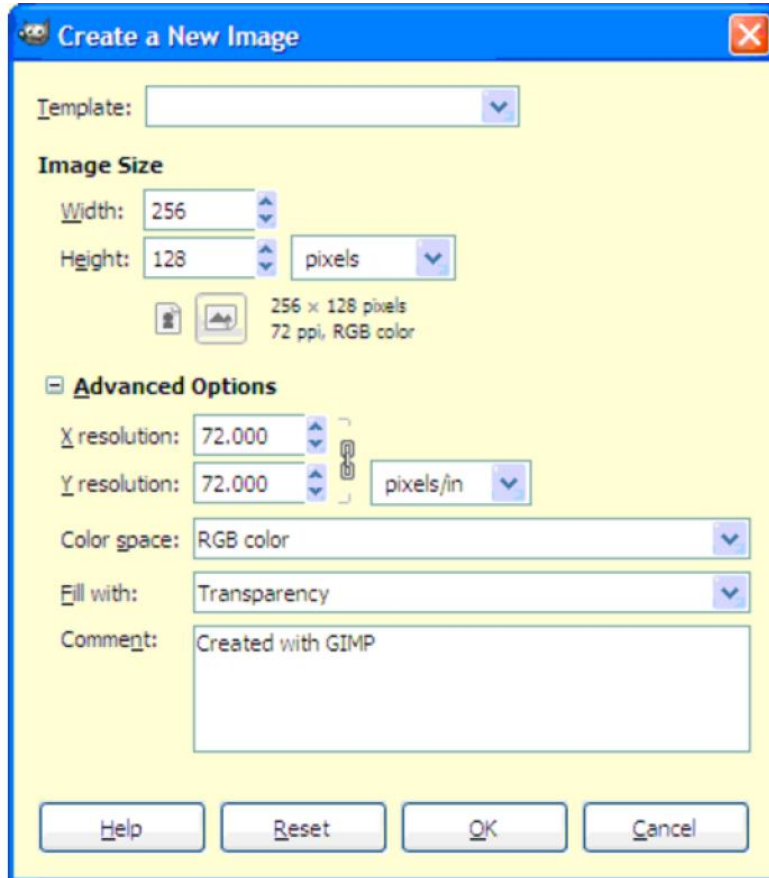


图 6-2 GIMP

2. 设置图像宽为 256 , 高 128. 这尺寸正好可以让我们的 tile sheet 容纳 4 行 32x32 的 tile.

3. 当你创建完图像后, 通过 Image -> Configure Grid 选项设置网格的大小为 32x32. 这一步完成后, 你需要确认 View 下拉菜单中的 Show grid 已经被勾选上. 如图 6-3 所示.

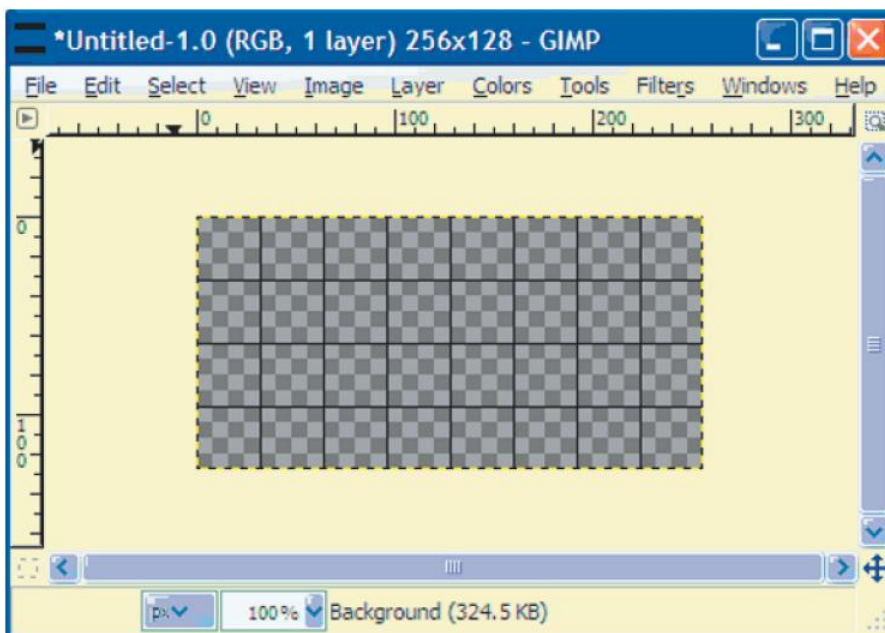




图:6-3

下面的步骤你可以挑选一些你找到的素材.但是在这里,我们从 maze/tankbrigade.bmp (从 Spritelib GPL)挑选一些 tile 然后将他们放到我们的 tile sheet 中.图 6-4 是一个缩放后的 sheet.

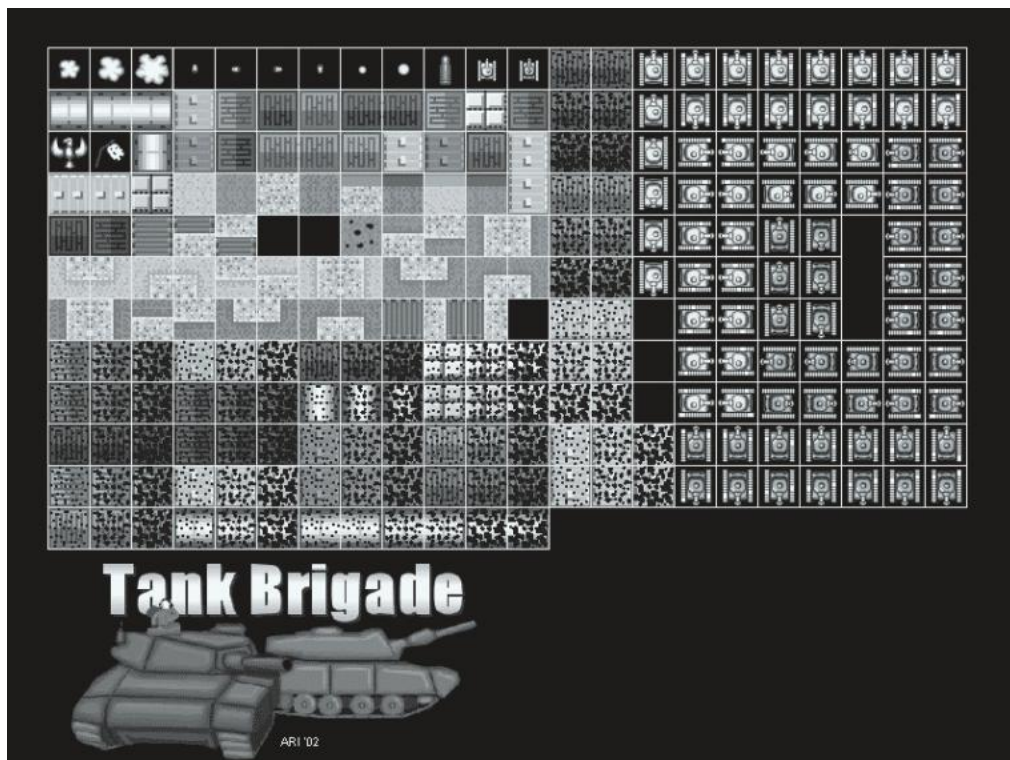


图 6-4

4 唯一的一个问题就是 BMP 格式没有背景透明属性.不过我们将粘贴过来的 tile 的背景颜色删除.还有一点要注意的就是 tankbrigade.bmp 文件中每个tile都是有边框的.设置选择框大小为 32x32,接着放大后再来选择我们需要的部分.如图 6-5

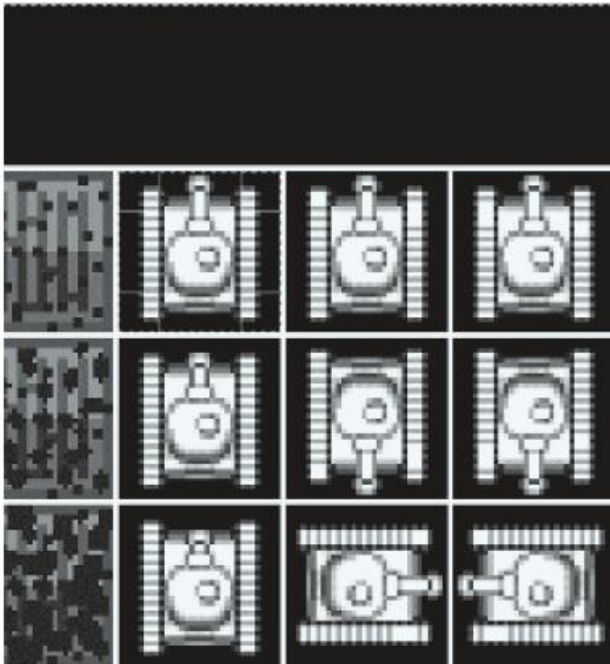


图 6-5 切割一个 32x32 的图像,注意网格之间的分割线.

5 当你选择好第一个要复制的图像后,你可以直接将其粘贴到我们刚刚创建的 sheet 中.然后一直重复这个步骤直到如图 6-6 中的 sheet 一样(注意我们已经将背景色去掉了)



图 6-6 完成的 tile sheet

如果你使用 Mappy 创建关卡(level)将第一个 tile 设置为非 sprite tile 即路或者墙 tile 就显得非常重要.我们将通过 2 个图层(layer)来创建关卡(level).背景图层将包含墙和路 tile, 前景图层将仅仅包含我们游戏中色 sprite.前景前景图层将覆盖到背景图层的上面.前景图层的所有 tile 不会包含内容为空的 sprite tile 或用 Mappy 导出时 id 不会为 0.当我们的代码解析 sprite 位置的数据时,它会忽略任何 id 为 0 的 tile.如果我们将在 sprite 中设置了 id 为 0 的 sprite,它永远也不会出现在游戏中.所以我们将非 sprite 的路或墙 tile 放置到 tile0 的位置上,当我们进行到 Mappy 部分时这一观点就会显现出来.



6. 你需要将 tile 中物体周围未使用到的背景颜色删除并将其设置为透明.最好的实现方法是使用橡皮擦设置为 1x1 像素,将图像放大到 800%并擦除图像周围深颜色的像素点.或使用其他的工具,例如 Fuzzy Select 工具也可以达到这个目的并且速度更快.在 FW 或 PS 中的 Magic

Wand 工具更适合完成这一工作.

7.将这个文件存储成名为 tank\_sheet.png 的文件.确定 Merge Visible Layers 选项已被选中并且在导出界面中的 Flatten Image 选项没有被选中.Flatten 将会设置图像的背景颜色为白色,Merge 将会保持背景透明.在 PS 中可以选择 Save For Web,FW 中可以通过 Save as 并选择 png 选项来轻松的完成.

## 用 Mappy 创建游戏关卡

对于开发者,所有武器中最有用的一个武器就是地图编辑器.我们非常喜欢用的一款地图编辑器名为 Mappy.Mappy 的基础班是免费的.但是我们强烈建议花费 19 美元来注册一个完整版.

更多关于 Mappy 的信息可以在(<http://www.tilemap.co.uk/>)站点看到.更多的功能和介绍已经超出了本书的范围.我们会展示通过 Mappy 创建关卡的每一个基础操作.下面我们将使用前面创建好的 tank\_sheet.png tile sheet.目前最新的版本 Mappy 需要一个额外的 DLL 才能使用 PNG 文件.所以你需要下载一个 Zip 文件将其中的 DLL 文件解压到 Mappy 文件夹中.

当你下载完后,解压,然后将其放到 Mappy 的文件夹中(没有官方的安装程序)你需要解压 libpng12.zip 文件(可以在 <http://www.tilemap.co.uk/pngfiles.html> 获取到),然后就可以在程序中使用 PNG 文件了.

### 现在我们来创建基于 tile 的关卡地图:

1.打开 Mappy.

2.在 File 下拉菜单中,选择 New Map 然后设置 tile 大小为 32x32,map 中 tile 为 20x15.坦克大战的关卡屏幕尺寸为 640x480, 即 宽:32x20 乘以 高:32x15

3.Mappy 会警告你需要导入一个 tile sheet 到程序中.在 File 菜单,选择 Import 然后选择我们用 GIMP 创建的 tank\_sheet.png 文件.

4.tile 会在右边显示.仔细观察 tile sheet,你就会注意到有一个空的 tile 插入到 sheet 的开始处.这是 tile 事实上并没有在 sheet 中,只是有助于快捷的删除 tile.这个细节在我们为 flash 导出数据时会被用到.

## 创建一个 双图层 map

我们将使用 Mappy 来创建一个双图层的关卡地图.第一层(layer 0 )将被用作背景图层.第二个图层(layer 1)将被用作盛放所有的 sprite 对象.

### 创建背景图层

这一图层将由路 tile(Mappy tile 1)和墙 tile(Mappy tile 25-32)组成.因为 Mappy 在 tile sheet 的开头处添加了一个橡皮 tile,所以你会发现 Mappy 中的 tile 数量比我们实际的数量多一个.不用担心,当我们从



Mappy 到处 tile map 时,我们会自动将每个 Mappy 中 tile 的 id 数减 1,所以数据和 sheet 中的 tile id 仍然是匹配的.

用路 tile 和墙 tile 创建一个如图 6-7 的 tile map

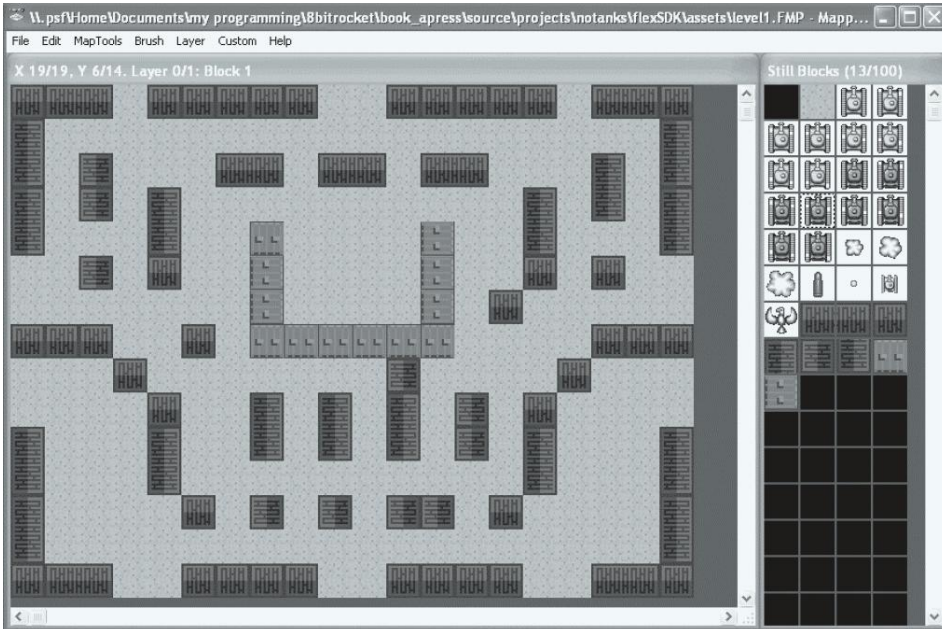


图 6-7 为 No Tanks! 创建的一个简单的关卡

为坦克大战设计一个关卡非常简单.事实上,唯一需要用多个 sprite 来表示的就是绿色坦克和那个代表必须被收集的宝物且看起来像金色的鸟这两个物体.灰色的 tile 代表坦克可以在上面移动的路,蓝色 tile 代表的是坦克无法直接穿越的墙.即:所有的 tile 种类是 TILE\_WALL,或是 TILE\_MOVE 否则就是 Sprite 动画组的一部分.

如果你将一个 路 tile(就是唯一的灰色路径)在一边将其部分放到边界外另一部分放到边界内,同时在另一边同样的对立位置(x 或者 y 相同)也放置一个 tile,我们的引擎就会为这两个 tile 建立一个连接通道.在其他的一些游戏引擎需要你手动来设定哪些 tile 作为连接通道.当任何路 tile 在对立的位置存在对应的路 tile 时,我们的引擎会默认为其创建连接通道.

## 创建 Sprite 图层

创建完背景图层后,我们接下来继续创建 sprite 图层:

- 1.在 Mappy 中,单击 Layer -> Add Layer 选项.然后就会在项目中添加 Layer 1.我们会在这一图层重新创建 sprite 组.

- 2.选择 Layer 下拉菜单中 Background Layers Darkened 选项中的 Onion Skin.当我们开始编辑 Layer 1 时这一步骤会带来很多好处.

- 3.在 Layer 下来菜单中,选择 Layer 1.这是你应该看到的是 Layer 在最顶层,变暗的背景 layer 在下面.如图 6-8 所示.

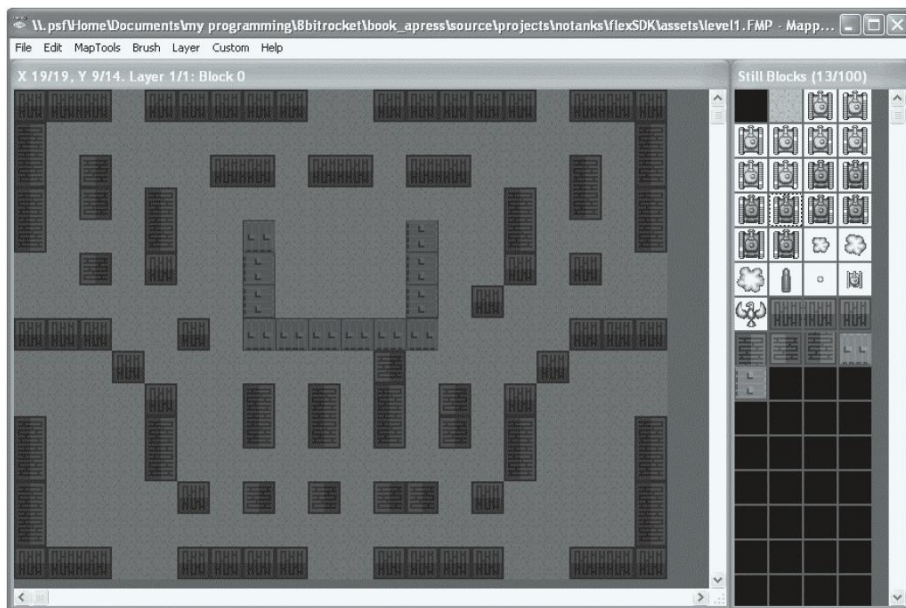
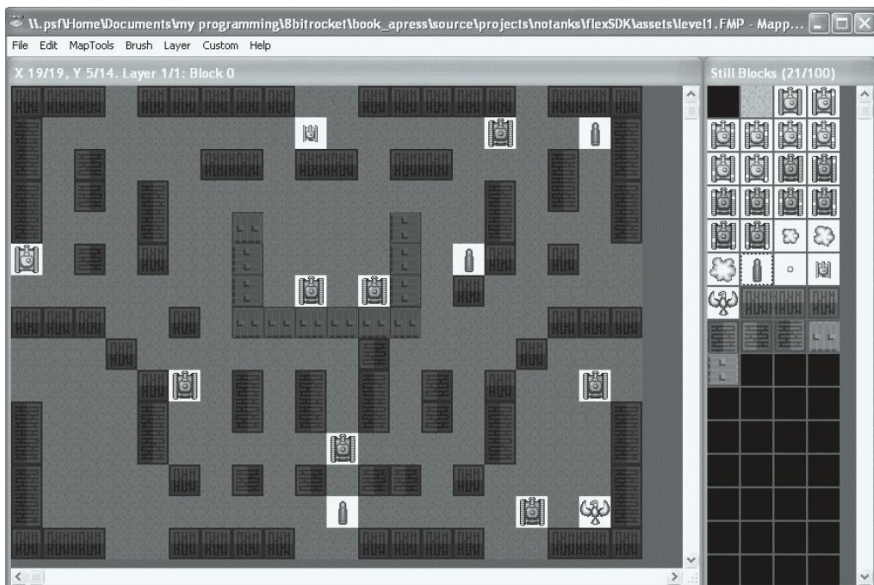


图 6-8 Sprite 图层(Layer 1) 是空的.Layer 0 是灰色的.

下面我们需要开始填充 Sprite 层.这一步很简单,如图 6-9 所示.



用 sprite 组填充的 Sprite 图层(Layer 1)

5. Sprite 图层已经完成了.现在我们将 2 个图层导入到我们的游戏.

将关卡数据保存为 FMP 文件

下一步就是将文件保存为 level1.FMP 文件.这一步将创建迷宫的数据和 tile sheet 一起保存到文件里,以后我们可以很方便的再次编辑或者更新.

从 Mappy 导出关卡.



我们将分别导出背景图层(Layer 0)和 Sprite 图层(Layer 1).在 Mappy 的 Custom 下拉菜单中有一个名为 Export For ActionScript 选项.选择这一选后,在我们导出时,会自动将当前选择的图层的信息保存为创建关卡地图所需的二维数组数据.

## 从 Mappy 导出背景图层.

下面是导出背景图层的步骤.

选择背景图层(在 layer 下拉菜单中选择).

2 在 Custom 下拉菜单中将 Export For Actionscript 选中.

3 接下来你会看到一个警告:当前图层将会被创建成一个平面地图;意思就是没有动画导出.我们并没有创建任何动画 tile,不仅如此 Mappy 还可以完成很多其他的事情,只是我们没有地方来写了.

4 单击 OK,文件类型为 as 文件.在这里,我们设置为 level1\_back.as.单击 Save.

5. 接下来,它会问你 map 的校正值是多少. 由于 Mappy 在 sheet 起始处添加了一个额外的空白 tile,所以现在 ID 会比我们在 Flash 中看到的 ID 大 1. 因此我们在输入框中键入-1,然后单击 OK 按钮.

这张 map 导出的 ActionScript 代码如下所示:

```
1. var map = [  
2.  [26,24,25,0,26,26,26,26,26,0,0,26,26,26,26,26,0,24,25,26],  
3.  [27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,27],  
4.  [29,0,28,0,0,0,24,25,0,24,25,0,24,25,0,0,0,27,0,29],  
5.  [27,0,28,0,27,0,0,0,0,0,0,0,0,0,0,27,0,29,0,27],  
6.  [29,0,0,0,29,0,0,30,0,0,0,0,31,0,0,29,0,0,0,29],  
7.  [0,0,28,0,26,0,0,31,0,0,0,0,31,0,0,26,0,26,0,0],  
8.  [0,0,0,0,0,0,31,0,0,0,0,31,0,26,0,0,0,0,0],  
9.  [26,26,26,0,0,26,0,30,30,30,30,30,30,0,0,0,26,26,26],  
10. [0,0,0,26,0,0,0,0,0,0,0,28,0,0,0,26,0,0,0],  
11. [0,0,0,0,26,0,0,27,0,27,0,27,0,28,0,26,0,0,0,0],  
12. [27,0,0,0,27,0,0,29,0,29,0,29,0,28,0,27,0,0,0,27],  
13. [29,0,0,0,29,0,0,0,0,0,0,0,0,0,0,29,0,0,0,29],  
14. [27,0,0,0,0,26,0,28,0,28,0,28,28,0,26,0,0,0,0,27],  
15. [29,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,29],  
16. [26,24,25,0,0,26,26,26,26,0,0,26,26,26,26,0,0,24,25,26]  
17. ]
```

## 导出 Sprite 图层

下面是导出 Sprite 图层(layer1)的步骤:

1.在 layer 下拉菜单中选中 Sprite 图层(layer 1)



2.选择 custom 下拉菜单中的 Export For ActionScript 选项.

3.看到如前面一样的警告,我们并没有创建动画 tile.

4.单击 OK,键入文件名.在这里我们输入 level1\_sprites.as.单击 save

5.接下来,同样的输入校正值为 -1

导出的ActionScript代码如下所示.注意图层中空白的地方都是用0来表示.我们会在下一章中编写如何布景,我们只需要知道 id 值是 0 的 tile 代表空白即可.

```
1. var map = [
2.  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
3.  [0,0,0,0,0,0,0,0,0,0,22,0,0,0,0,0,9,0,0,20,0],
4.  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
5.  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
6.  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
7.  [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,0,0,0,0,0],
8.  [0,0,0,0,0,0,0,0,0,0,9,0,9,0,0,0,0,0,0,0,0,0],
9.  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
10. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
11. [0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0,0,0,9,0],
12. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
13. [0,0,0,0,0,0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0],
14. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
15. [0,0,0,0,0,0,0,0,0,0,20,0,0,0,0,0,9,0,23,0],
16. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
17. ]]
```

## 创建项目

本书中所有的游戏,包括坦克大战,都会用到我们在第二章创建的框架.我们将分别在 Flash IDE 和 FLash Develop(FlexSDK)来创建包结构.

## 在 flash IDE 中创建坦克大战项目

下面是在 Flash IDE 中创建的步骤:

1.打开你的 Flash.我使用的是 CS3 , 这些步骤版本在 cs4 或 cs5 中应该是一样的.

2.在 /source/projects/notanks/flashIDE/文件夹 中创建一个名为 notanks 的 fla 文件.

3.在 /source/projects/notanks/flashIDE/文件夹 中为我们的游戏创建一个包结构:  
/com/efg/games/notanks/



4.设置 FLA 文件的 FPS ( 帧频 ) 属性为 30.舞台的宽为 640 像素 , 高为 500 像素。

5 设置文档类为 com.efg.games.notanks.Main

6.我们还没有创建 Main.as 类,所以你会看到警告.我们将在后面创建这个类.

7.下一步,将框架中可复用包中的类添加到 fla 文件中.首先在 发布设置( publish settings)中选择 FLash 选项卡,单击 ActionScript 3 旁边的设置(Settings)按钮.

8 单击浏览(Browse to Path)按钮,找到我们在第二章创建的 /source 文件夹.

9.最后选择类文件夹,单击选择按钮.现在我们可以使用 com.efg. framework 中的框架来开始创建我们的游戏啦.

下面是 FLash IDE 版本的文件夹结构

```
[source]
  [projects]
    [notanks]
      [flexIDE]
        [com]
          [efg]
            [games]
              [notanks]
```

## 在 Flash Develop 中创建坦克大战.

下面是创建步骤:

1.在 /source/projects/notanks/ 文件下创建名为 flexSDK 的文件夹(如果你还没有创建的话)

2.打开 Flash Develop,然后创建一个新项目.在创建界面选择 Flex 3 Project 并设置项目名字为 notanks.项目放在/source/projects/

notanks/flexSDK 文件中.项目包的结构应该是 com.efg.games.notanks.

如果 Flash Develop 没有自动创建文件夹,请确认 Create

Folder For Project 已经被选择.

点击 ok 按钮来创建项目.

3.点击 Project > Properties > Classpaths 菜单的 Add Classpath 按钮.

4. 下一步,找到我们刚刚创建的 /source 文件夹,选择其文件夹下所有的文件.

5.单击 ok 然后单击 Apply 按钮.



6. 在 Project > Properties > Classpaths 菜单子项中,设置帧频为 30, 宽为 640, 高为 500.

下面的是 Flex SDK 版本的文件夹结构(在 Flash Develop 中)

```
[source]
  [projects]
    [notanks]
      [flexSDK]
        [bin]
        [obj]
        [lib]
        [src]
      [com]
        [efg]
          [games]
            [notanks]
```

现在项目中已经有了基础的框架.现在我们来讨论几个关于框架结构和被重用的类框架代码的问题.

不管是 Flex Builder, Flash Builder, 或是 其他的任何 IDE,我们都要做的步骤就是用该工具创建一个新的项目, 并设置其默认的编译类.

下面是在 Flash Develop 和 Flash IDE 中的工作流程说明: FLash 开发的一种开发方法就是使用 Flash IDE 来组织结构和素材,然后用 Flash Develop 来编写代码.如果这是你的工作流程,你可能会更喜欢刚刚在 Flash IDE 中创建的包结构.

## 在 XML 中描述我们的 tile sheet

在我们用 ActionScript 读取 level1 的地图文件前,我们需要一个描述文件来告诉游戏引擎我们的 tile sheet 都包含哪些种类的 tile.我曾经用过的一种方法就是创建 XML 文件来描述 sheet 中每个存在的 tile 种类.我们将会用 Actionscript 类来完成这项工作.

## 创建 TilesheetDataXML.as 类

这个类是一个 XML 数据的容器就如同类中的静态变量一样.这种构建方法对于那些不需要展示但是包含了我们在游戏中需要用到的数据的解析很有用.

下面是 FLash IDE 中的结构:

```
/source/projects/notanks/flashIDE/com/efg/games/notanks/TilesheetDataXML.as
```

下面的是 Flex SDK 的(使用 Flash Develop):

```
/source/projects/notanks/flexSDK/src/com/efg/games/notanks/TilesheetDataXML.as
```

下面是这个类的全部代码.



```
1.  /**
2.   * ...
3.   * @author Jeff Fulton
4.   * @version 0.1
5.   * //file name TilesheetDataXML.as
6.   */
7.   package com.efg.games.notanks {
8.
9.       public class TilesheetDataXML {
10.           public static var XMLData:XML=
11.               <tilesheet>
12.               <tile id="0" name="road" type="walkable"></tile>
13.               <tile id="1" name="player" type="sprite"></tile>
14.               <tile id="2" name="player" type="sprite"></tile>
15.               <tile id="3" name="player" type="sprite"></tile>
16.               <tile id="4" name="player" type="sprite"></tile>
17.               <tile id="5" name="player" type="sprite"></tile>
18.               <tile id="6" name="player" type="sprite"></tile>
19.               <tile id="7" name="player" type="sprite"></tile>
20.               <tile id="8" name="player" type="sprite"></tile>
21.               <tile id="9" name="enemy" type="sprite"></tile>
22.               <tile id="10" name="enemy" type="sprite"></tile>
23.               <tile id="11" name="enemy" type="sprite"></tile>
24.               <tile id="12" name="enemy" type="sprite"></tile>
25.               <tile id="13" name="enemy" type="sprite"></tile>
26.               <tile id="14" name="enemy" type="sprite"></tile>
27.               <tile id="15" name="enemy" type="sprite"></tile>
28.               <tile id="16" name="enemy" type="sprite"></tile>
29.               <tile id="17" name="explode1" type="sprite"></tile>
30.               <tile id="18" name="explode2" type="sprite"></tile>
31.               <tile id="19" name="explode3" type="sprite"></tile>
32.               <tile id="20" name="ammo" type="sprite"></tile>
33.               <tile id="21" name="missile" type="sprite"></tile>
34.               <tile id="22" name="lives" type="sprite"></tile>
35.               <tile id="23" name="goal" type="sprite"></tile>
36.               <tile id="24" name="blueblock1" type="nonwalkable"></tile>
37.               <tile id="25" name="blueblock2" type="nonwalkable"></tile>
38.               <tile id="26" name="blueblock3" type="nonwalkable"></tile>
39.               <tile id="27" name="blueblock4" type="nonwalkable"></tile>
40.               <tile id="28" name="blueblock5" type="nonwalkable"></tile>
```



```
41. <tile id="29" name="blueblock6" type="nonwalkable"></tile>
42. <tile id="30" name="blueblock7" type="nonwalkable"></tile>
43. <tile id="31" name="blueblock8" type="nonwalkable"></tile>
44.
45. <smallexplode tiles="17,18,17"></smallexplode>
46. <largeexplode tiles="17,18,19,18,17"></largeexplode>
47. </tilesheet>;
48.     } // end class
49. } // end package
```

这个类的目的就是描述 sheet 中每个位置上的 tile 种类,用这种方法不管是创建或修改 tile sheet 都不会依赖于 tile sheet 中 tile 的硬编码定义(也就是我们在 xml 对其的描述).

## tile 属性

在 每个<tile></tile> 数据元素中,我们都定义了 3 个属性来描述 tile.

id:id 就是 tile 的编号,从 0 到 31.在 tile sheet 中有 32 个 tile.共四行,每行 8 个.这个数字并非是实际中的代码,但是却可以让游戏开发者和关卡设计师了解我们是如何用代码将 tile 在 tile sheet 中的位置转换到一维数组中.

**name:** 这个是用来表示当前 tile 是否是 Sprite 组的一部分.有一些 tile 不是 sprite 组,但是他们也仍然要有名字.这些仅仅是为游戏设计者和程序员做一个说明.在下一章中,我们创建的代码会根据不同的 name 属性实例化不同的自定义类.这会让关卡设计师设置玩家坦克在关卡中的位置变得十分简单.

**type:** 有三种不同的种类: sprite ,walkable(可行走的)和 nonwalkable(不可行走的).所有的墙 tile 必须设置为 nonwalkable;所有的路和通道 tile 必须设置为 walkable.其他的所有东西都应该设置为 sprite .

XML 中的后 2 个属性并不是从 Mappy 中获得的.

```
<smallexplode tiles="17,18,17"></smallexplode>
```

```
<largeexplode tiles="17,18,19,18,17"></largeexplode>
```

我们添加的这些属性.是用来描述游戏中 2 种爆炸动画是由哪些 tile 组合而成的.例如,smallexplode(小型爆炸) 动画将会播放 tile 17 , 然后是 tile 18 , 然后又是 tile 17

.我们会在下一节来具体的处理这两个动画.

## 代码是怎么工作的.

TilesheetDataXML.as 是一个全局类或者可以理解为 static 类型的类,他们永远不需要被实例化.由于我们的 XMLData 被定义为静态,这就意味着我们可以直接使用 XML 数据而不需要将其实例化.



为什么我们不直接使用一个 xml 文件然后来读取呢?在本书中我们偏向于不用 swf 读取外部文件.这种方式创建的游戏更容易在网上传播.因此,我们已经做了很多工作来保证最终的 swf 文件所需的素材都被内嵌即不需要调用外部文件.

## 在代码中使用 tile sheet 数据

我们将创建一些事例代码,在下一章中我们会再去丰富他们.事例代码将解析 TileSheetDataXML.as 中的 XML 数据,然后将解析数据存到一个可以用来描述 XML 文件的一维数组中.

## 创建 GameDemo.as 类

创建一个名为 GameDemo.as 的 ActionScript 类文件.在本章的剩余部分中,如果你使用 Flash IDE 你可能会将这个类设置为 notanks fla 文件的文档类.如果是在使用 Flex SDK 和 Flash Develop 你可能会将其设置为 常编译类( always-compile class ).对于其他任何开发工具,参照其说明文档,将其设置为项目设置 base, main,

或者是 文档(document) class.

下面是 FLash IDE 的文件结构:

```
/source/projects/notanks/flashIDE/com/efg/games/notanks/GameDemo.as
```

这个是 FLex SDK(使用 Flash Develop)的:

```
/source/projects/notanks/flexSDK/src/com/efg/games/notanks/GameDemo.as
```

这是 GameDemo.as 类的第一次迭代.我们会在下面的章节中逐步完善这个类.将下面的代码敲入文件中并存储;我们会在下一节中来讨论这些代码:

```
1. package com.efg.games.notanks
2. {
3.     import flash.display.Sprite;
4.     /**
5.      * ...
6.      * @author Jeff Fulton
7.      */
8.     public class GameDemo extends Sprite{
9.         public static const TILE_WALL:int = 0;
10.        public static const TILE_MOVE:int = 1;
11.        public static const SPRITE_PLAYER:int = 2;
12.        public static const SPRITE_GOAL:int = 3;
13.        public static const SPRITE_AMMO:int = 4;
14.        public static const SPRITE_ENEMY:int = 5;
```



```
15.     public static const SPRITE_EXPLODE:int = 6;
16.     public static const SPRITE_MISSILE:int = 7;
17.     public static const SPRITE_LIVES:int = 8;
18.     private var playerFrames:Array;
19.     private var enemyFrames:Array;
20.     private var explodeFrames:Array;
21.     private var tileSheetData:Array;
22.     private var missileTiles:Array=[];
23.     private var explodeSmallTiles:Array;
24.     private var explodeLargeTiles:Array;
25.     private var ammoFrame:int;
26.     private var livesFrame:int;
27.     private var goalFrame:int;
28.     public function GameDemo() {
29.         initTileSheetData();
30.         trace("tileSheetData.length=" + tileSheetData.length);
31.         trace("playerFrames.length=" + playerFrames.length);
32.         trace("enemyFrames.length=" + enemyFrames.length);
33.         trace("missileTiles.length=" + missileTiles.length);
34.     }
35.
36.     private function initTileSheetData():void {
37.         playerFrames = [];
38.         enemyFrames = [];
39.         tileSheetData = [];
40.         var numberToPush:int = 99;
41.         var tileXML:XML = TilesheetDataXML.XMLData;
42.         var numTiles:int = tileXML.tile.length();
43.         for (var tileNum:int = 0; tileNum < numTiles; tileNum++) {
44.             if (String(tileXML.tile[tileNum].@type) == "walkable") {
45.                 numberToPush = TILE_MOVE;
46.             }else if (String(tileXML.tile[tileNum].@type) == "nonwalkable") {
47.                 numberToPush = TILE_WALL;
48.             }else if (tileXML.tile[tileNum].@type == "sprite") {
49.                 switch(String(tileXML.tile[tileNum].@name)) {
50.                     case "player":
51.                         numberToPush = SPRITE_PLAYER;
52.                         playerFrames.push(tileNum);
53.                         break;
54.                     case "goal":
```



```
55.             numberToPush = SPRITE_GOAL;
56.             goalFrame = tileNum;
57.             break;
58.         case "ammo":
59.             numberToPush = SPRITE_AMMO;
60.             ammoFrame = tileNum;
61.             break;
62.         case "enemy":
63.             numberToPush = SPRITE_ENEMY;
64.             enemyFrames.push(tileNum);
65.             break;
66.         case "lives":
67.             numberToPush = SPRITE_LIVES;
68.             livesFrame = tileNum;
69.             break;
70.         case "missile":
71.             numberToPush = SPRITE_MISSILE;
72.             missileTiles.push(tileNum);
73.             break;
74.     }
75. }
76.     tileSheetData.push(numberToPush);
77. }
78.     explodeSmallTiles = tileXML.smallexplode.@tiles.split(",");
79.     explodeLargeTiles = tileXML.largeexplode.@tiles.split(",");
80. }
81. }
82. }
83.
```

## 读取 tile sheet 数据

GameDemo.as 类文件首先做的就是读取并解析 tile sheet 数据.在上一节中,我们已经看过了 TileSheetDataXML.as 文件的结构.在上面的代码中,我创建了 GameDemo.as 类文件的第一个版本,它将会被用来演示我们游戏的一些功能.这一切并非是在浪费,因为我们在下一章中只需做一点小小的更改就可以将代码重用.

## 常量

你应该会注意到我们在代码的开头处创建了一些常量.这些用整数来定义的常量是用来描述 tile 在 tile sheet 中的位置的.背景只有 2 中基本的 tile 种类: TILE\_WALL 和 TILE\_MOVE.他们将是我们的游戏背景中最



底层的组件.每个 tile 对应的属性只可能是这两种中的一种(TILE\_WALL 和 TILE\_MOVE),绝不可能 2 种都是.在 XML 中,如果一个 tile 的 type 属性值为 walkable,则它将被设定为 TILE\_WALL;如果 tile 的 type 属性值为 nonwalkable,则它将被设定为 TILE\_WALL.其他任何用来描述交互对象或者不是用来创建游戏当前关卡背景的对象在 XML 中的 type 值都将被设定为 sprite.依据这些 tile(type 为 sprite 的)在 XML 中的 name 属性值,他们将会被分类处理.

剩下的 7 个常量((SPRITE\_PLAYER 到 SPRITE\_LIVES)分别游戏中的不同的 sprite.游戏引擎将根据 XML 中 tile 的 name 属性值来划分其种类.

**Player:** 属性为 player 的 sprite 是玩家在游戏中要控制的对象.为玩家创建的 tile 数组将会模拟出坦克的移动动画.玩家的 sprite 看起来就像存在数组(playerFrames)的中的一连串单帧画面.如果 tile sheet 中的玩家 tile 都在正确的位置上.当我们用代码读取 tile sheet 的 XML 数据时,当它碰到一个 tile 的 type="sprite" 并且 name="player",它就会将其添加到这个数组中.

### Enemy(敌人):

所有的机器人都是用的同样的外观.敌人所有的 tile 将会被存储到 enemyFrames 数组中,它的功能和 player 数组一样.游戏引擎会假定 tile sheet 中的 tile 都在正确的位置上.用代码查找 type="sprite" 和 name="enemy" 的 tile,并将他们都存储到数组中(enemyFrames).

### Missile(弹药):

玩家和敌军都将使用同样的弹药 tile.用代码查找 type="sprite"和 name="missile"的 tile,当找到子弹 tile 时我们会将其存储到数组中(missileTiles).虽然子弹仅仅是用一个 tile 来表示,但是我们将它存储到数组是因为这样可以很容易的将其换成一个多帧动画.

### Explosion(爆炸):

有 3 类 tile 被定名为: type="sprite" 和 name="explode1",name="explode2",name="explode3".

这些 sprite 的处理方法和其他的 sprite 不同,我们允许游戏程序员随意的使用这 3 类 tile.我们有 2 个 array 存放了指定的 tile,他们组成了 2 种不同的爆炸效果.

```
1. private var explodeSmallTiles:Array;
2. private var explodeLargeTiles:Array;
3. They are both set by reading data from the XML:
4. explodeSmallTiles = tileXML.smallexplode.@tiles.split(",");
5.
6. explodeLargeTiles = tileXML.largeexplode.@tiles.split(",");
```

7. The data looks like this:
8. `<smallexplode tiles="17,18,17"> </smallexplode>`
9. `<largeexplode tiles="17,18,19,18,17"> </largeexplode>`

第一种爆炸致使用了 2 中爆炸 tile.我们通过播放 tile17, 然后播放 tile 18 最后是 tile 17 动画串来表示一个普通的爆炸动画.让我们再来看看 tile sheet;如图 6-10



图 6-10 tile sheet

如果我们连续播放 tile 17(最小的爆炸画面的 tile),tile18 然后是 tile17(然后播放空帧),就会得到一个流畅,相对简单的爆炸动画.这个爆炸式用来表示玩家或者敌人的子弹击中 TILE\_WALL 或者是 无伤害值的爆炸.

另一个更大的数组中的 tile 由这一顺序组成: 17, 18,19,18,和 17.这会创建一个更大,更长的动画.他们用来表示当玩家或者敌人的子弹将击中敌对坦克时和玩家坦克被摧毁.

我们将这些数据存储到 TilesheetDataXML 文件中是为了避免让我们的游戏存在硬编码数据.我们可以直接在代码中设置这些数组的值,但是如果这样我们游戏的复用性和可改性都会大大降低.

### initTileSheetData 函数

GameDemo 的构造函数会调用 initTileSheetData 函数 然后打印出函数中创建数组的长度.通过遍历 XML 的每个 tileXML 变量,根据不同的属性将其分配这些数组中.TilesheetDataXML.XMLData 是一个静态属性.目的就是创建一个名为 tileSheetData 的一维数组,并用其保存每个 tile 在 tile sheet 中的位置信息.

XML 数据在 AS3 中很容易操作.我们可以很轻松的遍历 XML 中的每个节点来确定它将被存储到那个数组中.我们首先确定 tile 的数量:

```
var numTiles:int = tileXML.tile.length();
```



相当简单,对吗?接着我们就像遍历数组一样遍历 XML 的每个 `<tile></tile>` 节点的数据.我们首先确定他是 walkable 还是 nonwalkable.如果属性值是二者之一,则将其属性值(walkable 或者 nonwalkable)存储到本地的 numberToPush 变量中.如果 tile 的 type 属性是 sprite,我们就必须对其 name 属性进行第二次检查.根据 name 的值,switch:case 会根据不同的值采取不同的行为,也就是将其分配到不同的数组中.并且将当前 tile 的属性存储到 numberToPush 中.在当前 tile 被解析完成后,我们循环的最后将当前 tile 的属性添加到 tileSheetData 数组中.

你以前已经看到过最后的两行的代码.这个是关于爆炸的数组.我们在前面的章节提起过这点.

## 测试运行:

不管是在 Flash IDE 或 Flash Develop 中,你都可以直接测试.对于 Flash

Develop 记得将 GameDemo.as 设置为常编译文件( always-compile file).在 Flash IDE 中,确定 GameDemo 为 game\_demo.fla 的文档类,然后测试运行.如图 6-11 所示

```
tileSheetData.length=32
playerFrames.length=8
enemyFrames.length=8
missileTiles.length=1
explodeSmallTiles.length=3
explodeLargeTiles.length=5
```

图 6-11 trace 面板输出信息

输出面板显示了当我们调用 initTileSheetData 函数之后,各个数组的长度.trace 的第一行显示的是 tileSheetData 数组的长度(32).接下来我们看到 playerFrames 和 enemyFrames 数组的长度都是 8.missileTiles 数组仅仅是由一个 tile 组成,explodeSmallTiles 和 explodeLargeTiles 数组长度 3 和 5.

## 显示关卡

我们的下一个任务就是解决关卡数据并在屏幕上显示出.这将会用到一些 tile sheet blitting(blitting 为块传输)技术,这一技术我们会在 "Tile sheet blitting 和 sprite"一节中详细讨论.我们会继续使用 GameDemo.as 文件并且添加一些新的代码.在这一部分对于 Flex SDK 和 Flash IDE 的代码会有一些不同,因为他们在处理嵌入资源的方法不同.



例如,当你在 FLash IDE 的库中嵌入一个 PNG 的位图,其基类是 BitmapData,但是当你在 flex 项目嵌入同样的资源时,其基类是 BitmapAsset 的一个实例(Bitmap 的子类).所以不同的处理方法产生了 2 中略微不同的代码.让我们开始吧.

## 组织你的代码

如果你是用 FLex SDK,你需要确认将所有原始素材都放到了 assets 文件夹中.目前我们还没有创建这个文件夹.所以我们现在来做吧.让我们再来看一下本地的包结构:

```
[source]
  [projects]
    [notanks]
      [flexSDK]
        [assets]
          tank_sheet.png
        [bin]
        [obj]
        [lib]
        [src]
      [com]
      [efg]
      [games]
      [notanks]
        GameDemo.as
```

注意我们已经在 /source/projects/flexSDK 文件夹下新建了一个名为 assets 的文件夹.对于我们的 FLex SDK 项目,该文件夹和/bin/, /obj/, /lib/, /src/是同级的. 我们把 tank\_sheet.png 也放到了 assets 文件夹中.在编译时,我们要确保其可以找到嵌入资源所在的文件夹,所以其路径的正确性是一件很重要的事情.

如果我们使用 FLash IDE 我们只需要在 fla 文件中嵌入 tank\_sheet.png 文件.我们会在下一节中具体讲解.

## 在 FLex 项目有添加 Library.as 类文件.

Flash IDE 不需要 Library.as 类文件.因为在设计时 fla 已经包含了一套标准的处理嵌入资源的方法.在 FLex SDK 项目中,在设计时期并没有针对于嵌入资源的类库.因此对于 flex 项目,我们需要在编译时期嵌入我们的素材资源.如果你是使用 FLash CS3(不包含 CS3) 以后的版本中也可以使用 Libray.as 类文件.

Library.as 的原理是:我们想要用 Flex 的嵌入方式去尽可能模拟 FLash IDE 库的样式.我们库将会由一些代表了嵌入素材的静态常量组成.你要需要创建并存储 Library.as 类并将其存放到 GameDemo.as 同文件夹下.



下面就是 library.as 在坦克中的包结构:

/source/projects/notanks/flexSDK/src/com/efg/games/notanks/Library.as

下面是 library 类的全部代码:

```
1. package com.efg.games.notanks
2. {
3.     import flash.display.Bitmap;
4.     import flash.display.BitmapData;
5.     /**
6.      * ...
7.      * @author Jeff Fulton
8.      */
9.     public class Library {
10.         [Embed(source='../assets/tanks_sheet.png')]
11.         public static const TankSheetPng:Class;
12.     }
13. }
```

这个类是非常简单的.他嵌入了 tanks\_sheet.png 文件并且将其导出类命名为 TankSheetPng.注意 PNG 文件的地址是相对地址.这是静态常量是十分重要的.我们将会在游戏代码中使用这个静态常量 (TankSheetPng)

## 添加库素材(Flash IDE)

如果你是使用 Flash IDE,你就需要导入 tanks\_sheet.png 到库中并且设置其导出类名.注意在 flex 中的嵌入类名和这里的名字是一样的.之所以保持名字的一致,是为了避免从一种技术到其他技术时带来的一些琐碎的事情.

## 在 flex 和 Flash IDE 中使用库素材.

现在我们的 Flash Developer/flex 或 Flash IDE 的库中意境有素材,我们将要准备去使用这些素材.作为一个简单的例子,在下一部分,对于 FLEX 我们将使用下面的代码将 TankSheetPng 类的传入 TileSheet 类.

```
/******* Flex *****
private var tileSheet:TileSheet= new TileSheet(new Library.TankSheetPng().bitmapData,
                                                tileWidth, tileHeight);
/******* End Flex *****
```

注意这段代码是创建一个 TankSheetPng 类的新实例然后将 bitmapData 属性直接传入 TileSheet 的构造函数.我们之所以这么做是因为 TileSheet 类的构造函数需要我们传入一个 BitmapData 实例,而非



bitmap.对于嵌入的 bitmap 型的素材,因为 FLex 只允许我们将其定义为一个 Bitmap 类(实际上是 BitmapAsset).为了获取 bitmapData,我们不得不通过点(.)语法来引用它.

相反的,在 Flash IDE 中,库中的 bitmap 型素材其基类就是 Bitmapdata.所以对于其构造函数或者是声明 TileSheet 类在 flash 中会更简单一些.

```
//***** Flash IDE *****  
//private var tileSheet:TileSheet = new TileSheet(new TankSheetPng(0,0), tileWidth,  
                                                    tileHeight);  
//***** End Flash IDE *****
```

最大的不同之处在于 tankSheetPng 需要传入 2 个参数,否则就有运行时错误发生.这些值可以是 tank\_sheet.png 文件的实际大小,

或者是 0,0.传入 0,0 也是可以正常通过的,所以你没必要去将 tilesheet 的大小硬编码到你的 GameDemo.as 类中.

## tile sheet 的位图传送(blitting)和 sprite 动画

我们将要为我们的框架创建下一个可复用类 TileSheet 类.这个类可以用来描述一个 tile sheet 或者其他显示对象的所有需要拷贝的单元属性.在本章和其他的某些章节中,这个类都起到了很重要的作用.

在我们开始动手写 tile sheet 类的代码前,我们先来定义下"tile sheet blitting",因为在此之后你会很经常的看到这些词,并且在本书的余下章节你经常会看到"blit"这个词.

### 定义"Blitting"

在 70 年代末到 90 年代初,Blitting 主要被用于 video game 和电脑游戏开发.大多数街机,家用系统和电脑都会使用到一些类型为 tile,sprite sheet 或者是角色(character)sheet 来制作他们的游戏.一些游戏例如 Atari 公司早期的游戏小行星(Asteroids)就没有使用 tile sheet.但是 Atari 的家用系统 (Atari 2600, 5200, 7800, 800,等) 都使用了形为 blitting 名为 Player Missile Graphics (PMGs).这些 PMGs 在内存中是一些可自由操作的显示对象,这样就可以用有限的 PMGs 来组合出整个游戏的背景.在早期的系统中这些对象的数据不是被描述为类型,而是用一个 2 进制对象(开或者关)内存映射的机械码来表示,并用他们创建一个单色的显示对象(多颜色在 Atari 7800 and Lynx games).

在 80 年代,一代一代的操作系统都使用了这个技术并且为他们取了不同的名字: BOBs, sprites, blitter objects, OBJs 等等.其中一些被固化到硬件芯片上,另一些可以在软件中创建.例如,Atari 的第一代 ST 机器就很快,但是他没有硬件的 sprites.这个系统机制对于制作游戏是非常好的,但是需要开发有一定经验,因为软件技术比专用硬件更适合来创建游戏的 sprites.当 Commodore Amiga 系统来临后,硬件 sprites 可以很快的被渲染到屏幕上.此时相对而言硬件就可以更轻松的创建一个高速的街机游戏.

Nintendo Entertainment System (NES)可能是 80 年代最流行的机器,他当时是一个功能强劲的机器因为他使用了大量的 tiles,tile sheets 和 blitting 技术.

[后面一部分在另一台电脑上 稍后补齐 缺得内容为:262 部分段落,234 第 1,2 大段]



这一部分主要介绍历史....不太影响阅读 ]

## 再谈 Flash

在 Flash 中,一个 Sprite 是一个内建类.Flash Player 引擎会自己渲染他们,所以当 Sprite 移动或者改变时你不需要做任何工作来刷新屏幕.和硬件 sprite 不同的是,软件 sprite 实际上每帧都会去重绘背景,并且这些功能并非由硬件提供.Flash 的 sprites 仅仅是取了和硬件 sprites 同样的名字并且它本质上是一个没有时间轴的 MovieClip.Flash 使用一种名为 screen invalidation 的技术来标记需要重绘的区域,因为在每一帧中,向量渲染引擎渲染需要重绘的区域比渲染全屏会更好.

如果你在浏览器中右键单击一个 Flash 影片,你会看到一个名为"显示重绘区域"的选项.点击这个选项,你就会看到 screen invalidation .在一个游戏中会用到很多显示对象,你会看到红色线框的无效区域遍布整个屏幕.如果在游戏中使用全屏的 blit 技术,你将会紧紧看到一个红色线框(或者是基于 blit canvas 使用数量的红色线框)

Flash 使用向量引擎来渲染所有 sprites 除非他们被设置为位图缓存(cached as a bitmap).如果一个 Sprite 被设置为位图缓存(Cache as Bitmap),当它移动时(或者因为某周原因而变形了),在变化过程中的每一帧,它都需要重新缓存,这就导致了处理器要做双倍的工作.当有很多对象在屏幕上移动或者变形时,Flash 向量引擎就需要不断地标记并重绘屏幕的绝大部分区域(由于向量被设置为位图缓存).在这种情况下,因为 flash 向量引擎并非效率卓越,我们将会探索一种使用 tiles sheets 和 blitting 的方法来最优化这一过程.

我们并非完全的忽视 Flash 向量引擎.在本游戏中使用 tile sheet,对于旋转和缩放你将会看到使用 Flash 中 sprite 带来的好处.我们游戏并不会包含数百个移动对象.它仅仅包含一些相关的移动或者相关的动画 sprite.对于我们游戏中的 sprite(例如坦克,弹药和爆炸),我们将会使用一个名为 blitting to individual display objects(译者注:义如其名针对某个显示对象的 blitting 技术).这就意味我们同时得到了 tile sheet 和 Flash 内建 sprite 对象的优点.

当我们创建坦克的 tile sheet 时,我们为每个坦克都放置了 8 帧.其每一帧都包含了在坦克履带上有细微差别的图片并以此来模仿其移动.我们并没有在 tile 中为这些 sprite 创建上下左右四个方向的不同版本,这样做是因为当我们将 tile sheet 的某个位图快传输到 Flash 的 sprite 中时,我们可以利用向量引擎的旋转功能来得到图像在该方向上的图像.

在 11 章中,我们将创建一个将位块渲染到单画布的游戏(全屏位块传输).这类的游戏中,我们将不能使用 Sprite 类和内建的向量引擎来旋转.在本章中,我们将会创建我们自己的系统来旋转位块传输对象.

## 了解 tile 和 Sprite sheets 的不同之处

你可能听说过 "tile sheet" 和 "sprite sheet" 这两个词可以交替使用.本质上讲他们是同一个东西: 一个在逻辑和物理上可以被分割为不同区域并且他们可以用来表示屏幕上的显示对象并且可以组成该对象的动画.一些基于 tile 的游戏使用一个由 tile 组成的 sheet 来渲染.他们其中的一些 sheet 可能用来表示 sprite 帧动画.在我们的游戏中,我们已经将所有的 tile 混合到同一个 sheet 中.但是这并不紧要.事实上,将他们合并到一起回更节省内存.



有很多种途径使用 tile sheets 来做基于位块传输的 flash 游戏.在本章中,我们将会使用其中的 2 中技术:

## 全屏位块传输

全屏位块传输是位块传输技术中最为极端的一个版本.理论上他可以带来很好的渲染速度,但是作为代价,其结构最为复杂并且程序步骤更为受限.它需要开发者使用一个 BitmapData 对象,我们称之为一个 canvas.所有的对象都通过 copyPixels 操作被放置在这个 canvas 中.除了这个 Bitmap 对象没有任何 sprites, MovieClips, Bitmaps 或其他对象被添加到 Flash 的显示列表中.这种位块传输操作需要开发者在对象显示到屏幕前,使用 tile sheets 或 bitmapData 数组提前渲染好所有的显示对象和动画.

## 独立对象的位块传输

独立对象的位块传输用于多现实对象渲染的游戏,他们每一个都有一个用来显示 tile sheet 或者是数组中某个基于 Bitmapdata 的 canvas 对象.例如,在 Flash 中我们从 tile sheet 中 copyPixels 出一个 32x32 的 tile 并将其设置为一个 bitmap 的 bitmapData 属性.如果我们将这个 Bitmap 添加到一个 Sprite 对象中并且设置其 x 于 (width\*.5)\*-1, y 于 (height\*.5)\*-1.然后我们就有了一个 32x32 且注册点在中心的 tile.现在我们可以任意的旋转,缩放,移动这个 Sprite.如果我们想要改变它的 tile,我们只需将新的数据复制到该 Bitmap 的 BitmapData 属性即可.我们这样做是为了让我们的游戏同时拥有 tile sheets 和 Flash Sprite 的优点.

## 融合不同种类的位块传输方法

我发现将 2 中不同的位块传输方法放到一起使用是非常灵活的.我已经分别用针对独立物体的位块传输和,单画布的方法分别实现了游戏.2 种方法都可以很好的运行,但是我们在这章和下一章中所制作的游戏将同时使用这两个技术.我们将使用一个和屏幕等尺寸的 canvas 来填充不同等级的地图.这就意味着这个 canvas 只包含 tile 属性为 TILE\_WALL 和 TILE\_MOVE.之后我们将要创建一个名为"BlitSprite"的自定义类来代表游戏中的 Sprites.这样结构会给我们带来很大的灵活性,除了 TileSheet 转换到 blit 动画 再转换到 sprite 这一过程较为复杂.这个类将继承扩展 Sprite 类,但是我们的转换顺序是 TileSheet 类到 blit 动画 到 sprite???.这一结构带来的好处是我们可以很方便的使用 Sprite 类的 rotation 属性,从而不需要去预渲染好坦克(每个都由 8 帧组成)在各个角度上的图像.

## 测试位块传输( blitting)的渲染速度

有很多人曾经问过我为什么我们喜欢使用位块传输( blitting )来操作动画和屏幕渲染.他们指出这一操作相对基于时间轴的动画而言更难,而且要花更多的时间在准备工作和代码编写上.在 2008 年,我们决定在我们的网站上用大量的渲染测试来向那些反驳者证明.在这里我们并没有列出所有的测试细节,但是我们会列出结果,这一结果会证明:相对基于时间轴的方法,使用一些 blitting(独立个体或者全屏)会带来很大的速度,帧频提升.



每一个测试都是不断向屏幕上添加对象同时计算平均帧频.在测试中,我分别测试了 100, 500, 1,000, 5,000, 10,000,

和 15,000 在屏幕上的情况.所有的测试都是在 Flash IDE 中完成并设置其帧频为 120.

## 基于时间轴方法的测试

这个测试使用的都是独立的 MovieClip 对象.我将每个 tile 放在 MovieClip 的每一帧中;如表:

MovieClip 数量	平均帧频
100 MovieClips	107 FPS
500 MovieClips	22 FPS
1,000 MovieClips	9 FPS
5,000 MovieClips	小于 1 FPS
10,000 MovieClips	未测试
15,000 MovieClips	未测试

## 独立个体对象的 blit 测试

这个测试使用的是独立的 Sprite.将 tile sheet 中代表动画每帧的 Bitmap 分别放入一个 Sprite. 我使用 BitmapData 的 copyPixels 方法从 tile sheet 上复制出对应每帧的 Bitmap 对象.

表 6-2. 独立个体对象的 blit 测试结果

Sprite 数量	平均帧频
100	124 FPS
500	124 FPS
1,000	91 FPS
5,000	18 FPS
10,000	9 FPS
15,000	6 FPS

## 测试全屏 sprite 位块传输(blit)

这一测试紧紧使用了一个 BitmapData 显示画布(canvas).BitmapData 对象的 copyPixels 方法用来从 tile sheet 中复制出由 tile 组成的动画所需要的数据.结果如表 6-3 所示

表 6-3



---

Sprite 数量	平均帧频
100	126 FPS
500	124 FPS
1,000	93 FPS
5,000	22 FPS
10,000	12 FPS
15,000	8 FPS

## 理解这些测试结果

在我们的游戏中,我们至少要保持 30FPS.正如你看到的,使用基于时间轴方法,当舞台上 有 100 个对象时 可以有一个很好的帧频,但是当数量到达 500 时,帧频就大幅下降了.但是 当我们使用 独立个体 sprite 和全 屏 blit 方法时,至少可与容纳 1000 个物体.使用全屏 blit 添加 5000 个对象时(30FPS 低一点)的帧频和使用 基于时间轴方法添加 500 个对象的帧频是相同的.

这意味着什么?这意味着只要我们需要我们可以从 Flash player 那里榨取到更高的效率.并不是所有的 游戏都需要这么高的效率,在本书中我们将会展示怎样将 blitting 和 tile sheet 方法应用到绝大多数游戏中. 伴随着手持设备和移动设备上基于 Flash player 的游戏出现,我们可以预示到怎样从 Flash player 中榨取出 更高的效率将会变得越来越重要.

如果你对新版本 player 的渲染速度测试感兴趣,本书的第 6 章代码包含了这些代码和必要的文件.fp10 对这 3 中渲染技术都做了很大的改善.

## TileSheet 类

TileSheet 类是 blitting 操作的核心.它包含了在 Library 中 tile sheet 的的引用和 blitting 相关操作需 要的的信息.这个类将一直存在于我们游戏的框架中.这个类还将应用于本书的其他的一些游戏中.

创建如下的包结构并创建 TileSheet.as 文件:

```
/source/classes/com/efg/framework/TileSheet.as
```

我们先来看一下类的代码:

```
1. package com.efg.framework{
2.     import flash.display.BitmapData;import flash.geom.*;
3.     /**
4.      * ...
5.      * @author Jeff Fulton
6.      */
7.     public class TileSheet {
8.         public var sourceBitmapData:BitmapData;
9.         public var width:int;
10.        public var height:int;
```



```
11.     public var tileWidth:int;
12.     public var tileHeight:int;
13.     public var tilesPerRow:int;
14.     public function TileSheet(sourceBitmapData:BitmapData,tileWidth:int,
15.         tileHeight:int ) {
16.         this.sourceBitmapData = sourceBitmapData;
17.         width = sourceBitmapData.width;
18.         height = sourceBitmapData.height;
19.         this.tileHeight = tileHeight;
20.         this.tileWidth = tileWidth;
21.         tilesPerRow = int(width / this.tileWidth);
22.     }
23. }
24. }
```

## 理解 TileSheet 的属性

为了可以将 tilesheet 作为游戏的图形资源,我们需要定义一些变量并且读取一些变量.构造函数包含三个信息:

```
1. (sourceBitmapData:BitmapData,tilewidth:int, tileheight:int)
2. private var sourceBitmapData:BitmapData;
```

在构造函数中你会发现这样一行代码:

```
3. this.sourceBitmapData=sourceBitmapdata;
```

sourceBitmapData 代表了实际 PNG sheet 中用于做 blitting 操作的 tile.这个变量持有着有构造函数中传入的 BitmapData 实例引用.在我们的游戏中,此游戏中使用的 sheet 尺寸是 256 128.这个尺寸正好可以容纳下 8 行 4 列大小为 32x32 的 tile.另两个变量代表了 sheet 的实际宽和高.

```
4. private var width:int;
5. private var height:int;
```

在构造函数中,这些变量将会被设置为传入的 BitmapData 的宽和高.

```
6. width = sourceBitmapData.width;
7. height = sourceBitmapData.height;
```

tileWidth 和 tileHeight 的值将直接被传入构造函数.

```
8. private var tileWidth:int;
9. private var tileHeight:int;
```

在构造函数中,他们将被赋值.



```
10. this.tileHeight = tileHeight;
11. this.tileWidth = tileWidth;
```

tilesPerRow 变量对于 blitting 操作非常重要.它被用来计算盛放 tile 的二维数组的行列值.例如 tilesheet 有 9 个 tile,当 tilesPerRow 的值为 8 时,你就会发现:我们必须在 tile sheet 的第二行才可以找到它.

```
private var tilesPerRow:int;
```

这就是 TileSheet 类的所有.下面我们来看一下等级数据中的 Level1.as 类.

## 读取等级信息

就像 tile sheet 的数据一样,所有的等级信息数据都将写入到一个类中.我们这样做是为了将游戏全部融入一个 swf 文件,游戏制作更简单.在坦克大战中有一个非常基础的基类,它包含了等级信息,我们称之为 Level.让我们来开始创建这个类吧.

## Level 类

Level 类是游戏中所有等级的积累.结构如下:

The Flash IDE:

```
/source/projects/notanks/flashIDE/com/efg/games/notanks/Level.as
```

The Flex SDK (using Flash Develop)

```
/source/projects/notanks/flexSDK/src/com/efg/games/notanks/Level.as
```

下面是其完整代码:

```
12. package com.efg.games.notanks
13. {
14. /**
15. * ...
16. * @author Jeff Fulton
17. */
18.     public class Level {
19.         public var backGroundMap:Array;
20.         public var spriteMap:Array;
21.         public var backGroundTile:int;
22.         public var enemyIntelligence:int;
23.         public var enemyShotSpeed:Number;
24.         public var ammoPickUp:int;
25.         public function Level() {
26.             }
27.     }
```



这个简单的基类仅仅定义了一些在每个游戏等级中都需要用到的信息.让我们来看看每个变量.

```
1. public var backGroundMap:Array;
```

backGroundMap 二维数组将用于存储我们在 Mappy 中为游戏创建的背景图层(layer 0)的数据.

```
1. public var spriteMap:Array;
```

spriteMap 二维数组将用于存储我们在 Mappy 中为游戏创建的背景图层(layer 1)的数据. ???2 个 for 不知道翻译的是否正确,觉得别扭

```
1. public var backGroundTile:int;
```

当绘制 sprite 图层时,backGroundTile 变量用来告诉游戏引擎 tile sheet 上的那些 tile 应该忽略掉.一般来说,这个 tile 是 0. 如果你回顾一下前面的内容,在 sprite 图层中,Mappy 将 id 为 0 的 tile 作为空白 tile. 我们在游戏中要用到 0, 但是如果你使用的是不同的编辑器,你可能要修改这个值.

```
1. public var enemyIntelligence:int;
```

enemyIntelligence 是一个在 0-100 的值.它被用来告诉游戏引擎敌军坦克做非追捕 AI 规则(下一章中)的随机移动的概率.该数值越高对军越聪明(???我觉得这里又问,前后意思正好相反)

```
1. public var enemyShotSpeed:Number;
```

enemyShotSpeed 是敌军弹药在每一个帧中移动的像素量.该数值越大敌军的子弹的射击速度越快.

ammoPickUp 变量表示当玩家拾取弹药时增加的弹药量.

## 创建 Level1 类

存储等级一数据的类名为 Level1.as.这个文件将是坦克大战包结构中的一部分.它是 Level 类的一个子类.

Flash IDE 中的路径:

[source][projects][notanks][flashIDE][com][efg][games][notanks]Level1.as

And here's the one for the Flex SDK (using Flash Develop):

[source][projects][notanks][flexSDK][src][com][efg][games] [notanks]Level1.as

下面是 Level1.as 的全部代码:

```
1. package com.efg.games.notanks
2. {
3.     /**
4.      * ...
5.      * @author Jeff Fulton
6.      */
7.     public class Level1 extends Level {
8.         public function Level1() {
```



```
9.         backGroundTile = 23;
10.        enemyIntelligence = 60;
11.        enemyShotSpeed = 2;
12.        ammoPickUp = 20;
13.        backGroundMap = [
14. [26,24,25,0,26,26,26,26,26,0,0,26,26,26,26,0,24,25,26],
15. [27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,27],
16. [29,0,28,0,0,0,24,25,0,24,25,0,24,25,0,0,0,27,0,29],
17. [27,0,28,0,27,0,0,0,0,0,0,0,0,0,27,0,29,0,27],
18. [29,0,0,0,29,0,0,30,0,0,0,0,31,0,0,29,0,0,0,29],
19. [0,0,28,0,26,0,0,31,0,0,0,0,31,0,0,26,0,26,0,0],
20. [0,0,0,0,0,0,0,31,0,0,0,0,31,0,26,0,0,0,0,0],
21. [26,26,26,0,0,26,0,30,30,30,30,30,30,0,0,0,26,26,26],
22. [0,0,0,26,0,0,0,0,0,0,0,28,0,0,0,0,26,0,0,0],
23. [0,0,0,0,26,0,0,27,0,27,0,27,0,28,0,26,0,0,0,0],
24. [27,0,0,0,27,0,0,29,0,29,0,29,0,28,0,27,0,0,0,27],
25. [29,0,0,0,29,0,0,0,0,0,0,0,0,0,0,29,0,0,0,29],
26. [27,0,0,0,0,26,0,28,0,28,0,28,28,0,26,0,0,0,0,27],
27. [29,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,29],
28. [26,24,25,0,0,26,26,26,26,0,0,26,26,26,26,0,0,24,25,26]
29. ];
30.
31.        spriteMap = [
32. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
33. [0,0,0,0,0,0,0,0,0,22,0,0,0,0,0,9,0,0,20,0],
34. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
35. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
36. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
37. [1,0,0,0,0,0,0,0,0,0,0,0,0,0,20,0,0,0,0,0],
38. [0,0,0,0,0,0,0,0,0,9,0,9,0,0,0,0,0,0,0,0],
39. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
40. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
41. [0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0,9,0,0],
42. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
43. [0,0,0,0,0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0],
44. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
45. [0,0,0,0,0,0,0,0,0,20,0,0,0,0,9,0,23,0,0],
46. [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
47. ];
48.    }
```



```
49.    }  
50. }
```

在类中我们可以看到,我们仅仅是将类中的变量填充上对应的值(注意 `Level1` 继承了 `level` ).我们还复制了从 `Mappy` 导出到 `level1_back.as` 中的 `backGroundMap` 二维数组的数据.我们将这个数据赋值给 `spriteMap` 数组变量

## 更新 GameDemo.as 文件

现在我们已经把 `tile sheet` 创建完毕了,将其放入库中,某个类持有其实例,关卡信息可以用来提供绘制,我们可以先把它放到屏幕上看看.要达到这个目的,我们要回顾下 `GameDemo.as` 文件并用 `Level1.as` 文件中的 `tile` 来编写一些用来绘制背景的代码.

为了最终将背景绘制到屏幕上,下面列出了必须要完成的 3 个步骤:

- 1.我们必须解析关卡数据并将其放入二维数组中.
- 2.我们必须将存放背景 `tile` 的二维数组绘制到我们的显示层上.
- 3.我们必须将图层添加到舞台上

## 导入的类

将 `GameDemo.as` 导入部分的代码全部替换成以下代码:

```
1. package com.efg.games.notanks  
2. {  
3.     import flash.display.Sprite;  
4.     import flash.display.Bitmap;  
5.     import flash.display.BitmapData  
6.     import flash.geom.Rectangle;  
7.     import flash.geom.Point;  
8.     import com.efg.framework.TileSheet;
```

注意我们为 `bitting` 操作添加了一些新类.`geom` 和 `display` 类包会完成绝大部分的工作.还有一点需要注意我们已经将 前面创建并存入框架中的 `com.efg.framework.TileSheet` 类导入了进来.

## 变量的定义

将 `GameDemo.as` 原来定义变量的部分全部替换为以下代码:

```
1. public class GameDemo extends Sprite{  
2.     public static const TILE_WALL:int = 0;  
3.     public static const TILE_MOVE:int = 1
```



```
4.      public static const SPRITE_PLAYER:int = 2;
5.      public static const SPRITE_GOAL:int = 3;
6.      public static const SPRITE_AMMO:int = 4;
7.      public static const SPRITE_ENEMY:int = 5;
8.      public static const SPRITE_EXPLODE:int = 6;
9.      public static const SPRITE_MISSILE:int = 7;
10.     public static const SPRITE_LIVES:int = 8;
11.
12.     private var playerFrames:Array;
13.     private var enemyFrames:Array;
14.     private var explodeFrames:Array;
15.     private var tileSheetData:Array;
16.
17.     private var missileTiles:Array=[];
18.     private var explodeSmallTiles:Array;
19.     private var explodeLargeTiles:Array;
20.     private var ammoFrame:int;
21.     private var livesFrame:int;
22.     private var goalFrame:int;
23.
24.     //the map definition
25.     private var tileWidth:int = 32;
26.     private var tileHeight:int = 32;
27.     private var mapRowCount:int = 15;
28.     private var mapColumnCount:int = 20;
29.
30.     //levels
31.     private var level:int = 1;
32.     private var levelTileMap:Array;
33.     private var levelData:Level;
34.     private var levels:Array = [undefined,new Level1()];
35.
36.     //full screen blit
37.     private var canvasBitmapData:BitmapData=new BitmapData(tileWidth * É
38.         mapColumnCount, tileHeight * mapRowCount, true, 0x00000000);
39.     private var canvasBitmap:Bitmap = new Bitmap(canvasBitmapData);
40.
41.     private var blitPoint:Point = new Point();
42.     private var tileBlitRectangle:Rectangle = new Rectangle(0, 0, É
43.         tileWidth, tileHeight);
```



```
44.  
45.    //***** Flex *****  
46.    private var tileSheet:TileSheet= new TileSheet(new É  
47.        Library.TankSheetPng().bitmapData,tileWidth, tileHeight);  
48.    //***** End Flex *****  
49.  
50.    //***** Flash IDE *****  
51.    //private var tileSheet:TileSheet = new TileSheet(new É  
52.        TankSheetPng(0,0), tileWidth, tileHeight);  
53.    //***** End Flash IDE *****
```

如果你使用的是 FLash IDE,你可能会想将 FLex 部分注释掉,然后将代表 tilesheet 实例的代码去除注释.

让我们先快速的浏览一下这些变量.我们将快速的了解一下这些变量的具体用处,当然我们只是简单的讨论一下并不会让你觉得晦涩.

这些是定义 map 的变量

```
1.  //the map definition  
2.    private var tileWidth:int = 32;  
3.    private var tileHeight:int = 32;  
4.    private var mapRowCount:int = 15;  
5.    private var mapColumnCount:int = 20;
```

这些变量定义了游戏屏幕中 tile 的宽和高.tileWidth \*mapColumnCount 等于用来绘制背景图层的 BitmapData 的宽(32x20 = 640).

tileHeight \* mapRowCount 等于用来绘制背景图层的 BitmapData 的高(32x15 = 480).

这些变量是关卡信息:

```
1.    private var level:int = 1;  
2.    private var levelTileMap:Array;  
3.    private var levelData:Level;  
4.    private var levels:Array = [undefined,new Level1()];
```

这些变量被用来告诉游戏引擎当前正在显示哪个关卡并决定将数据存储到哪个数组中.levelTileMap 将保存 Level1.as 的 tile.levelData 变量将会持有一个 Level 子类实例(如 Level1)的引用.levels 数组保存着所有 Level 派生出的子类.目前我们的游戏只有 Level1.as 文件.注意我们将 Level1 数组中检索值为 0 的元素设置为 undefined.由于我们游戏中并没有 Level 0 , 所以我们就它当做一个占位符放到数组中.当前的关卡数据将通过如下的方法获取:

```
1.    levelData=levels[level];
```

下面都是用于全屏 blit( full-screen blit)的变量



```
1. //full screen blit
2.     private var canvasBitmapData:BitmapData=new BitmapData(tileWidth *
3.         mapColumnCount, tileHeight * mapRowCount, true, 0x00000000);
4.     private var canvasBitmap:Bitmap = new Bitmap(canvasBitmapData);
5.
6.     private var blitPoint:Point = new Point();
7.     private var tileBlitRectangle:Rectangle = new Rectangle(0, 0, tileWidth,
8.         tileHeight);
9.
10.    //***** Flex *****
11.    private var tileSheet:TileSheet= new TileSheet(new
12.        Library.TankSheetPng().bitmapData,tileWidth, tileHeight);
13.    //***** End Flex *****
14.
15.    //***** Flash IDE *****
16.    //private var tileSheet:TileSheet = new TileSheet(new
17.        TankSheetPng(0,0), tileWidth, tileHeight);
18.    //***** End Flash IDE *****
```

这些变量在 Blitting 操作显示关卡背景 tile 时会被用到。canvasBitmapData 定义了一个用来存储游戏中 tile 尺寸为 20x15, 属性为 TILE\_MOVE 和 TILE\_WALL 的 BitmapData 对象。这就是我们所说的全屏 blit 图层 (full-screen blit canvas)。canvasBitmap 是实际被添加到舞台上的显示对象。

blitPoint 是 tile 在 tile sheet 上的起点位置, 以该点为起点并通过 copyPixels 将 tile 绘制到 canvasBitmap 上。tileBlitRectangle 是 copyPixels 从起点开始复制一个大小为 32x32 的矩形。在前面我们已经看到过 tileSheet 变量; 我们通过传入在库中的 TankSheetPng 实例和 tile 的宽高来实例化 tileSheet。

## 构造函数

新的构造函数添加了一些新代码来调用一些新的函数。

```
1. public function GameDemo() {
2.     initTileSheetData();
3.     readBackGroundData();
4.     readSpriteData();
5.     drawLevelBackGround();
6.     addChild(canvasBitmap);
7. }
```



首先我们从 Level 实例中读取背景数据.接着我们调用 readSpriteData 函数.在本章中我们不会去实现这个函数,但是我们需要指出在背景数据未读取完毕之前 sprite 数据是不会开始读取的.接下来我们通过调用 drawLevelBackGround 函数将背景绘制到 canvasBitmapData 上,最终将 canvasBitmap 添加到舞台上.

readBackGroundData 函数是一个全新的函数.我们将通过它读取关卡数据并创建二维数组 levelTileMap.

```
1. private function readBackGroundData():void {  
2.     levelTileMap = [];  
3.     levelData = levels[level];  
4.     levelTileMap = levelData.backGroundMap;  
5. }
```

readBackGroundData 函数非常简单.在第 7 章中,我们将添加更多的代码从 Level 类的实例中读取更多的关卡信息,但是现在,我继续使用这个简单的类.首先我们通过[]来快速初始化 levelTileMap 数组.接着,我们将代表当前关卡的 Level 类实例的引用存入到 levelData 数组中.最后,我们用 levelTileMap 引用 levelData.backGroundMap 数组.这就是我们使用关卡数据之前所要做的工作.

### readSpriteData 函数

readSpriteData 函数目前只是一个空函数,我们将在第 7 章将其实现.

```
1. private function readSpriteData():void {  
2.     //place holder for reading sprite data and placing sprites on the screen  
3. }
```

### drawLevelBackGround 函数

正如在本章中已经讨论过的,drawLevelBackGround 函数是全屏位块传输(blit)操作的核心.下面是该函数的完整代码,我们稍后就会对其逻辑进行详尽的解说.

```
1. private function drawLevelBackGround():void {  
2.     canvasBitmapData.lock();  
3.     var blitTile:int;  
4.     for (var rowCtr:int=0;rowCtr<mapRowCount;rowCtr++) {  
5.         for (var colCtr:int = 0; colCtr < mapColumnCount; colCtr++) {  
6.             blitTile = levelTileMap[rowCtr][colCtr];  
7.  
8.             tileBlitRectangle.x = int(blitTile % tileSheet.tilesPerRow) * tileWidth;  
9.             tileBlitRectangle.y = int(blitTile / tileSheet.tilesPerRow) * tileHeight;  
10.  
11.             blitPoint.x=colCtr*tileHeight;  
12.             blitPoint.y = rowCtr * tileWidth;
```



```
13.  
14.         canvasBitmapData.copyPixels(tileSheet.sourceBitmapData,  
15.         tileBlitRectangle, blitPoint);  
16.     }  
17. }  
18. canvasBitmapData.unlock();  
19. }
```

## 组织游戏的渲染

为坦克移动设计的关卡背景将由属性为墙和路的 tile 组成,背景将使用全屏位块传输(full-screen blit)方法并将其放置在一个名为 canvasBitmapData 的 BitmapData 图层上.为了可以看到我们的 tile,canvasBitmap 将会被添加到显示列表中.独立的 tile 将会通过 copyPixels 被放置到 canvasBitmapData 上,这一操作会将每个 tile 都绘制到 canvasBitmapData 层而非绘制到独立 sprite 或者 MovieClip 上.这一方法和一些广为流传的方法相悖,例如:

将一个 movieClip 的二维数组添加到屏幕上并且使用 gotoAndStop 来选择要播放的帧.

使用存在于屏幕 Bitmap(甚至是 tile sheet)2 维网格中独立的 Sprite(或者 Bitmap)对象来显示每个 tile.???

这个函数被用来位块传输背景 canvasBitmapData 的不同 tile(可行走和不可行走的 tile).下面是这个变量的定义:

```
1. private var canvasBitmapData:BitmapData = new BitmapData(tilewidth * mapcols,  
    tileheight * maprows, true, 0x00000000);
```

他是的大小游戏屏幕大小(640 x480)一样.它有一个填充色为黑色但透明的背景(你看不到颜色是因为它是完全的透明).当然我们也可以选择任何颜色,但是我们更喜欢使用 0x00000000 来初始化所有背景透明 BitmapData.一连串的 8 个 0 也同时反应出改对象是一个空图层.如果如果我们需要一个黑色图层,我们需要将改 32 位数字的前两位改为 FF.例如该数字为 0xFF000000 则背景将是一个不透明的黑色背景.当游戏进行时游戏背景图层时不会改变的.所有它只需被复制到 canvasBitmapData 一次就可以了.

代码使用了嵌套循环的结构,第一层嵌套将遍历数组 levelTileMap 的每一列.第二层嵌套则是遍历数组的行.记住:这里的坐标系和常规的坐标系相反的 [x-axis], [y-axis] .我们之所以这样做是因为在二维数组中使用[行][列]的更方便.

让我们看看怎么样从 tile sheet 复制数据到 canvasBitmapData.下面的几行代码是本章中的最重要信息的一部分

```
1. blitTile = levelTileMap[rowCtr][colCtr];  
2. tileBlitRectangle.x = int(blitTile % tileSheet.tilesPerRow) * tileWidth;  
3. tileBlitRectangle.y = int(blitTile / tileSheet.tilesPerRow) * tileHeight;
```



```
4. blitPoint.x=colCtr*tileHeight;
5. blitPoint.y = rowCtr * tileWidth;
6. canvasBitmapData.copyPixels(tileSheet.sourceBitmapData,tileBlitRectangle, blitPoint);
```

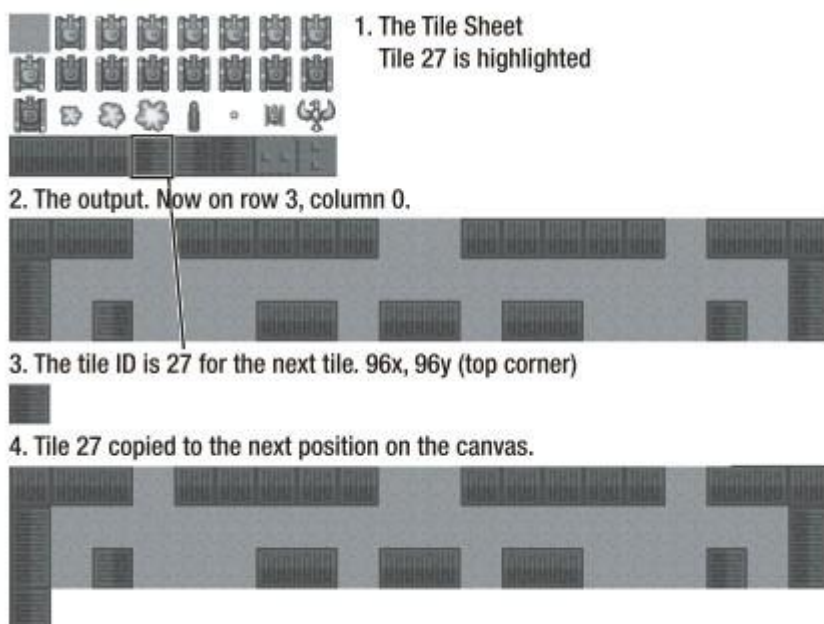
第一行是通过 levelTileMap[rowCtr][colCtr]的方式获取 tile 对应的 id,并将其复制给 blitTile 变量.id 值就是该 tile 在一维中的下标.

tileBlitRectangle 是 tile sheet 中 tile 的大小.

接下来的 2 行使用使用了一点数学方法来计算出 tile 在 tilesheet 上的起始点坐标.由于 sheet 实际上是 2D 的,并且我们的数组仅仅是一维的,所以我们需要这个技巧

起始的 x 坐标是通过模运算并四舍五入获得的,而 y 坐标则是通过除法运算四舍五入得到的.

如图 6-12 所示:



那么,6-12 究竟代表了什么?

- 1.我们看到 drawLevelBackground 函数在执行过程中,它依据 levelTileMap 中的二维数组中数据,从 tile sheet 中复制了 2 行数据到 canvasBitmapData
- 2.下一个要取的 tile 是第一个 tile(tile 0),它在第 4 行(row3):levelTilemap[3][0] 行的值为 3,并且列的值为 0.
- 3.blitTile 变量用来盛放 levelTilemap[3][0],这个 tile 是 tile sheet 中的第 27 个 tile:

tileBlitRectangle.x=int(27 % 8)\*32 or x= 3\*32 = 96

tileBlitRectangle.y=int(27 / 8)\*32 or y=3\*32 = 96

复制起始点从 tile sheet 中的 96x, 96y 开始.这两个值相等只是一种巧合.



blitPoint.x 和 blitPoint.y 的值为 tile 宽高分别乘以该 tile 在 tile sheet 中的二维行列值;如图 6-13

最后,我们从 tile sheet 中复制出像素点到背景图层上( background canvas):

```
backgroundBD.copyPixels(tilesheet.getSourceBitmap(),blitrectTile, blitpoint);
```

复制源文件是一个我们在 library 中创建 tile sheet 类的实例对象.他是一个 32x32 矩形(tileBlitRectangle) ,并且其位置(blitPoint)我们在前面已经设置过了.

我们为关卡中的每一个 tile 都进行这一操作.

优化!在这里我们通过使用一个 单例 Point 类实例和一个单例 Rectangle 类实例来完成一个最大的优化操作.创建对象是一个非常消耗处理器的密集活动.通过使用这两个类的单例实例,虽然我们将为此付出一点点内存但是节省的处理器资源消耗却是可观的.在下一章中,我们将会把这些技术应用到 blit 显示对象的每一帧上,很快就可以看到这些技巧了.

最后一个你可能会注意到的是:在 copyPixels 操作前有 canvasBitmapData.lock ,在 copyPixels 之后有 canvasBitmapData.unlock .这一操作有益于屏幕渲染.它允许 copyPixels 关闭屏幕.lock 操作会告诉所有将这一 BitmapData 对象当做属性的显示对象,在 unlock 操作之前不要更新他们自己.这一操作将会让屏幕的刷新尽可能的平滑.

这是本章中最后的一段代码.如果你将他们全部键入,没有敲错的话 用 Flash Develop/Flex 测试运行,你会看到如图 6-13 所示的输出.



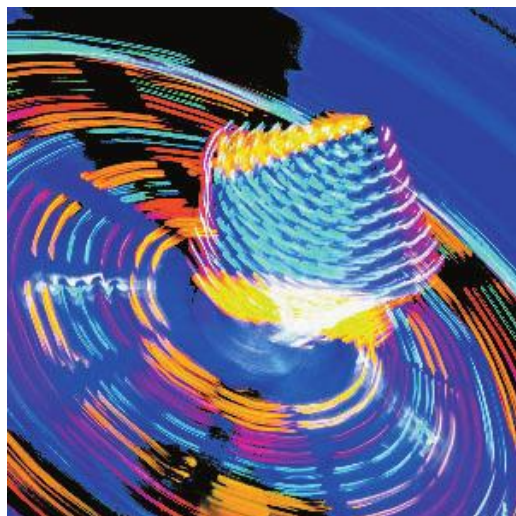


图 6-13

## 总结:

在本章中我们接触了很多东西.你已经开始创建你自己的 tile sheet,使用你选择关卡编辑器创建了 2 图层的关卡.接着我们创建了一个 TileSheet,Library 和 Level 基类及其子类 Level1。新的 GameDemo.as 将使用这些来完成 No Tanks 的基础部分.游戏使用了全屏的 blit 技术.从这种技术出发我们又讨论了多种 blitting 显示技术并且看到了一些渲染对比结果.

在本章的结尾,我们已经有了一个可以完整运行的 No Tanks! 游戏.继续阅读吧,接下来的内容是关于基于 tile 的迷宫/ai 和 blitting.



## 第七章

# 构建坦克大战游戏

---

在前一章中,我们讲了许多关于坦克大战游戏的内容。我们涉及到了块结构( tile sheets ),级别,XML,和关于位图复制的东西。在这一章中,我们将完成游戏并涵盖更多的高级的主题,像基于块的网格平滑运动还有一个“足够好的”敌人追逐 AI。我们将很快的熬过成堆的代码,我们也需要更有效率的完成游戏的所有逻辑部分,还要根据情况更改框架。所以,坚持住,仔细的阅读所有的细节。还有,如果你在 AS3 的语言概念上卡住了,而书中跳过或介绍的太简短,就请查翻阅 Adobe 出品的 AS3 帮助文档。

我们将必须在剩余的时间里快速的讲解许多的知识以完成这个游戏。这样,将会在一些知识点更详尽的讲解,而其他的部分就仅仅展示代码或做一个简短的介绍。

- 为游戏对象创建 BlitSprite 类
- 为基于迷宫的块结构逻辑扩展 BlitSprite 类
- 用位图块给一个 Sprite 做动画
- 在用户设计的游戏关卡中移动玩家和敌人
- 用区域和逻辑的思路创建基本的追逐 AI
- 用直线开火对抗敌人玩家



- 用 look-ahead 变量做碰撞检测和运动
- 加上声音和场景完成游戏
- 为坦克大战！修改 framework
- 扩展类库

上一章我们创建了一个叫做 GameDemo.as 的文件。我们将在接下来的内容中继续修改这个文件直到我们完成游戏。你要做的第一件事是在 Flex,Flash Develop,或 Flash IDE( 旁边的 no\_tanks fla 是为 IDE 准备的)中打开 GameDemo.as 文件。我们最终会把 GameDemo.as 的游戏文件名字更改为 NoTanks.as, 并把它整合到 framework 中去, 但是现在, 你可以先不去管它。现在你已经打开了这个文件, 走, 我们去干活吧。

在这一章中, 我们将用一种独特的方法来创建最终的游戏代码。我们从第六章中完成的 GameDemo.as 文件开始,再视情况添加一些代码片段, 就成了另一个文件。我们称为通过复用来搭建游戏。在我们完成游戏的最后部分前会有六段复用。每次复用的结果就是一个可以编译并查看游戏进展最新版本的类文件。在我们复用之前, 我们先以一些可以在 2D 的基于块的网格构架移动游戏角色的理论开始。

## 基于块的移动理论

有许多可以在场景中让游戏角色运动的方法。就像你在第六章中看到的, 我们用 32\*32 像素的块创建了一个游戏场景布局, 这些游戏角色将会在这个场景中对抗。

### 块的跳跃

区块跳跃一些早期游戏的方法, 它们试图创建基于区块的游戏。这些游戏中的所有角色从在网格中一个块跳跃到另一个块, 而不是在块之间平滑的运动。当然他们并不是真正的跳跃, 而是他们的移动被限制在每个块中的中心点。这样, 每次用户点击方向键, 屏幕上的角色会简单直接的精确地走到下一个区块中 (一次 32 像素)。两个位置之间并没有流畅的运动, 在玩家看来好像是直接跳到下一个格子中了。不能够在两个区块位置中心之间移动。除非动画做的特别号, 否则这种类型的运动看起来不舒服而且控制性不太好。相对这种基于区块的运动类型, 另一种经典的吃豆人游戏能够流畅的在屏幕上的每一个像素点之间平滑的运动。无论怎样, 如果游戏运用块到块的跳跃方案, 就会实际上解决两个典型的游戏编程问题:

- 系统资源管理: Flash Player (和更早的 8 位系统) 和系统运行时创建模拟街机游戏币真正的快速-动作游戏有更多限制。
- 基于网格的运动和逻辑: 一些基于网格的运动很难管理 (特别是拐角转弯), 但是如果游戏角色的运动和逻辑都在块的中心点, 那么创建起来比较简单。



我们会给我们的坦克大战游戏使用一个更运动更平滑的块到块的类型，更像吃豆人的那种。即使最新版的 Flash Player 和 AS3 可以相当快的渲染场景和执行游戏逻辑，我们还是会使用一些我们自己的优化，以确保游戏更好的运行。此外，系统方面我们会使用块到块的平滑类型运动，让用户和玩家有更好的体验，还允许我们使用整合了基于网格的块到块型运动的数学和逻辑。

## 块到块的平滑型运动

块到块的平滑性运动解决了通常由于关联了跳跃型游戏带来的视觉上太丑陋的问题，还保留了使用基于网格运动的区块中心运动带来的编程便利性。这让游戏看起来可以在网格上自由运动，游戏角色将会移动一组精确的像素数直到碰到下一个中心点。这个系统的限制并没有看起来那么多，比如可以让角色在任何时间反相运动，即便在到达下一个中心点之前也可以。我们不是简单的使用中间动画来显示块之间的角色运动，因为这不允许角色在到达下一个块之前进行交互操作，会带来一个不舒适的用户体验。

我们会以第六章完成的主要角色开始，编写它的运动。如果你还没有读完第六章，那么你接着去读完并到此书官网上下载上章的最终文件会比较好。

## BlitSprite 类

当前的 GameDemo.as 在区块结构和关卡数据中读取并将关卡显示到场景中。到现在为止其实根本就没有游戏，但是也花了我们一些时间。我们现在需要的是一个把我们的视觉元素放置到场景中并用区块控制他们的样貌的方法。这个类精确地控制视觉元件，能够实现区块的动画，还包含能够控制基本运动的所需属性。

我们会把这个创建的类作为我们可重用的始于第二章的框架的一部分。这个文件在框架中的只位置应该是：

```
/source/classes/com/efg/framework/BlitSprite.as
```

下面是这个类的所有代码：

```
■ package com.efg.framework
■ {
■     import flash.display.Bitmap;
■     import flash.display.BitmapData;
■     import flash.display.Sprite;
■     import flash.geom.Point;
■     import flash.geom.Rectangle;
■
■     /**
■      * ...
■      * @author Jeff Fulton
■      */
■
■     public class BlitSprite extends Sprite {
■         private var bitmap:Bitmap;
```



```
■ public var bitmapData:BitmapData;
■ private var tileSheet:TileSheet;
■ private var rect:Rectangle;
■ private var point:Point;
■ public var animationDelay:int=3;
■ public var animationCount:int=0;
■ public var animationLoop:Boolean=false;
■ public var tileList:Array;
■ public var currentTile:int;
■ public var tileWidth:Number;
■ public var tileHeight:Number;
■ public var nextX:Number=0;
■ public var nextY:Number = 0;
■ public var dx:Number=0;
■ public var dy:Number = 0;
■ private var doCopyPixels:Boolean = false;
■ public var loopCounter:int = 0;
■ // counts the number of animation loops if useCounter is set to true;
■ public var useLoopCounter:Boolean = false;
■
■ public function BlitSprite(tileSheet:TileSheet, tileList:Array, firstFrame:int)
{
■     this.tileSheet = tileSheet;
■     tileWidth = tileSheet.tileWidth;
■     tileHeight = tileSheet.tileHeight;
■     this.tileList = tileList;
■
■     rect = new Rectangle(0, 0, tileWidth, tileHeight);
■     point = new Point(0, 0);
■
■     bitmapData = new BitmapData(tileWidth, tileHeight, true, 0x00000000);
■     bitmap = new Bitmap(bitmapData);
■     bitmap.x = -.5 * tileWidth;
■     bitmap.y = -.5 * tileHeight;
■     addChild(bitmap);
■     currentTile = firstFrame;
■     renderCurrentTile(true );
■ }
■
■ public function updateCurrentTile():void {
```



```
■
■     if (animationLoop) {
■
■         if (animationCount > animationDelay) {
■             animationCount = 0;
■             currentTile++;
■             doCopyPixels = true;
■
■             if (currentTile > tileList.length - 1) {
■                 currentTile = 0;
■                 if (useLoopCounter) loopCounter++;
■             }
■
■         }
■         animationCount++;
■     }
■ }
■ public function renderCurrentTile(forceCopyPixels:Boolean=false):void {
■
■     if (forceCopyPixels) {
■         doCopyPixels = true;
■     }
■
■     if (doCopyPixels) {
■         bitmap.bitmapData.lock();
■
■         rect.x=int(tileList[currentTile] % tileSheet.tilesPerRow)*tileWidth;
■         rect.y=int(tileList[currentTile] / tileSheet.tilesPerRow)*tileHeight;
■
■         bitmap.bitmapData.copyPixels(tileSheet.sourceBitmapData, rect, point);
■         bitmap.bitmapData.unlock();
■     }
■     doCopyPixels = false;
■ }
■
■ public function dispose():void {
■     bitmap.bitmapData.dispose();
■     bitmap = null;
■     rect = null;
```



```
■      point = null;
■      tileList = null;
■      }
■      }
■      }
```

BlitSprite 类是坦克大战中所有角色的基石。它是对内建的 Sprite 类的扩展，所以我们可以方便的利用它的旋转之类的其他功能。它没有时间轴，所以我们用自己特别的方法来做角色动画，通过附有一组区块 id 的 TileSheet 类的实例，而第一个块的 id 显示一张块 id 的列表。这些块的 id 会在我们第六章创建的 8\*4 的区块上关联其他区块。这些区块会通过 copyPixels 方法，在 BlitSprite 实例中应用到 Bitmap 对象。

## 用 blitSprite 类做动画

- 没有足够的篇幅在这展示 BlitSprite 类的所有细节，下面这些是要点：
- BlitSprite 类需要 TileSheet 的实例。它会被储藏在一个叫做 tileSheet 的私有变量中。
- 还需要一个数组的实例，表示区块的 id 列表（一个数组中的一维列表），列表表示它的动画的帧序列。这些放在 tileList 数组变量中。
- 需要一个帧值（第一帧），用来设置 currentTile 的公共属性。
- 它包含了一个 Bitmap 对象（叫做 bitmap），放在 Sprite 的初始位置（ $-0.5 * width$ ,  $-0.5 * height$ ）。这让 BlitSprite 能够在自身旋转的时候在它的中心轴旋转 Bitmap。
- Bitmap 包含一个 BitmapData 的实例（叫做 bitmapData），会被用来做位图复制。每当 BlitSprite 改变成一个新块是，这些像素就会从 TileSheet 拷贝到 BitmapData。
- 坦克大战的类会随着帧频调用 BlitSprite 类的 updateCurrentTile 函数
- animationLoop 布尔值控制着 BlitSprite 是否需要为 Sprite 进行寻找新块的循环工作。玩家和敌坦克都不想在他们不可移动时调用各自的运动动画，所以在那个实例中，这个变量会被设置为 false。
- animationCount 用来计算帧，直到这个帧数比 animationDelay 的值更大为止。如果更大了，currentTile 的值就更新到 tileList 数组中的下一个值。在这种情况下，doCopyPixels 的公共属性就会设置为 true。
- renderCurrentTile 函数用来复制一组新的像素到 Bitmap 对象里的 BitmapData 中去。坦克大战游戏类会随着每次帧运动调用此函数。如果为 true，它将强迫函数从区块中复制当前块。
- 接着，doCopyPixels 布尔值已被赋值。是在 updateCurrentTile 函数中设置的。如果它为真，区块像素会被从 tilesheet 复制到 bitmap 实例里的 bitmapData。
- 我们在 copyPixels 和 bitmapData.unlock()之前设置 bitmapData.lock。这是一种加快渲染速度的优化。本质上，当你这么做的时候，你是在告诉所有的对象将 BitmapData 实例作为属性参考，以在设



置 unlock 前不要更新他们自己。这将会让动画更圆润，也允许处理器在执行复制时不用必须保持在所有对象上的 copyPixels 操作。

- 这个对象使用后在场景中被删除，然后 dispose 函数用来清理的内存。

## TileByTielBlitSprite 类

我们所有的游戏对象都会用 BlitSprite 类作为基类，尽管他们中有的会使用其他的但是是 BlieSprite 类的子类的类。TileByTielBlitSprite 类很简单的扩展了（含有一些必需的能够以逻辑方式移动格子的变量的）BlitSprite 类。

我们会创建这个类作为可重用框架的一部分。这个文件的位置在：

/source/classes/com/efg/framework/TileByTileBlitSprite.as

这是此类的完整代码：

```
■ package com.efg.framework
■ {
■     /**
■     * ...
■     * @author Jeff Fulton
■     */
■     public class TileByTileBlitSprite extends BlitSprite {
■
■         public var inTunnel:Boolean = false;
■         public var currRow:int;
■         public var currCol:int;
■         public var nextRow:int;
■         public var nextCol:int;
■
■         public var moveDirectionsToTest:Array = [];
■
■         public var missileDelay:Number = 100;
■         public var missileTime:int;
■         public var healthPoints:int;
■
■         public var distinationX:Number;
■         public var distinationY:Number;
■         public var currDirection:int = 0;
■
■         public var moveSpeed:Number = 2;
```



```

■      public var currentRegion:int;
■
■      public function TileByTileBlitSprite(tileSheet:TileSheet, tileList:Array, #
■          firstFrame:int) {
■          super(tileSheet, tileList, firstFrame);
■      }
■  }
■  }
■  }

```

TileByTileBlitSprite 类会是 BlitSprite 类的一很好的一部分，但是通过创建其他的类，我们可以使用 BlitSprite 类为非块基础的游戏服务，而且还没有那轻微过头的额外的（将会是非必须的）变量。我还给这个类添加了 healthPoints 变量，因为只有玩家和敌坦克有此需要。我们会对所有这些变量的使用进行测试，就像我们路过本章。

## 在地图中移动玩家角色

我们将会为玩家创建一个运动系统，次系统允许关卡设计师有足够的自由而不必担心他们应该如何摆放这些块。就是说我们会有一些为创建关卡而预设的原则。只要是关卡包含 TILE\_WALL 和 TILE\_MOVE 块，还有一个绿色的 Player 坦克，一个 Goal，就不会出现任何问题。我们不会再给关卡设计设置其他任何规则。这就像是被一个平台游戏引擎设计师雇佣一样~。

多数的追逐/迷宫类型的经典版本都是用这种方法创建的，就是 预设数学和逻辑 被用来设置 玩家或别的角色 在游戏中的十字路口 可能会去的方向。这个被硬编码的信息用来简化原本可能需要的让角色右转的数学，并约束了角色在十字路口上的自由方向运动。如果你在 Google 搜索 Java 或 JavaScript 版本的吃豆人源码就会看到这种类型的逻辑代码。我们希望我们的游戏引擎可以扩展尽可能多的不同类型的关卡，所以我们应该已经选择了一个不同的，更流畅的方法来给编写我们的运动逻辑。

## 中心块的技巧

自从我们准备用一种不同的方法做我们的游戏，而不是过去的吃豆人模式，更不希望是创建一个固定地图关卡然后预设或硬编码原本自由的运动数据类型，我们打算设置一些与运动基本规律并让游戏引擎对关卡地图做出动态的回应。我们会用一个技巧来做这些。这个技巧总是在角色走到块的中心时计算运动的逻辑。这种理念对于方形的块和角色视觉元素块和同样大小的块很适用。这听起来好像有很多限制，但是这个技巧被证明能解决基于格的运动的很多问题。

例如，如果你曾玩过吃豆人，你会注意到，除非在那个方向上有一个开放的块（空间），否则角色就不能转向。这个非常重要的理念为这个游戏的运动方式和逻辑设置了一个基础的规则。我不知道 Toru Iwatani(原版吃豆人的中途开发者)的运动是怎么编写的，但是我们知道如何去模仿怎样做出来。吃豆人的角色不能移动到一个装不下它整个身体的空间里去，但是它不需要或使用像素级精确地碰撞检测。它用基于块的碰撞检测。这个游戏逻辑仅仅取药知道下一个块是什么类型的然后计算角色能不能移动到这个块里去。在角色不处于当前块的中心时，游戏不需检查下一个块，因为这样做会返回非正值或负值，就像角色移动到了一个开阔的处于块块之间的地方，或角色同时在两个或多个块上面。



我们的游戏会更进一步，强制我们的角色停在每一个块的中心。你会看到当我们开始编写代码和测试游戏元素时，如果在按下方向上的块是一个 TILE\_WALK 块，角色就会根据方向键移动到相应方向。角色随着帧开始运动并继续运动直到走到了下一个块的中心在停下来。这并没有听起来那么多的限制，因为如果用户按住方向键（并且玩家想去的方向上是有空间的），玩家会继续不停顿的走到这个开放空间里去。这个运动包括转向和直走。反转方向会更细腻圆润，同时玩家可以在任何时候反转方向，甚至在到达下个块的中心之前。这个吃豆人有些不同，吃豆人只有在撞墙或拐角才会停止。

做这种类型运动逻辑有一个告诫，就是角色每帧可移动的像素数必须能被块尺寸整除。所以，如果我们有一个 32\*32 的块，我们的角色每次可以动 1, 2, 4, 8, 16 或 32 像素。如果不是这样，他们或许就会永远也无法真正到达一个块的中心点，而且逻辑也会失败。

## 添加角色（复用 1）

我们现在在 GameDemo.as 文件中开始通过简单的修改代码把玩家添加到场景中去，此 .as 文件是我们游戏的第一个基石。这个角色会是 TileByTileBlitSprite 的实例。让我们开始（通过添加一些代码到我们的所有所需变量的变量定义区）来为玩家操控一个角色和运动。

我们会创建一个叫做 GameDemoIteration1.as 的单独的文件。

如果用 FlashIDE，路径是这里：

```
■ /source/projects/notanks/flashIDE/com/efg/games/notanks/GameDemoIteration1.as
```

如果是 Flex SDK（用 FlashDevelop），路径是这里：

```
■ /source/projects/notanks/flexSDK/src/com/efg/games/notanks/GameDemoIteration1.as
```

## 给 复用 1 改类名

这个游戏示范有一个新的文件名，现在需要改变类名以匹配。我们同时也需要改变构造器函数的名字，如下：

首先，我们改变新类名。

```
■ public class GameDemoIteration1 extends Sprite
■ {
```

然后，更改构造器函数

```
■ public function GameDemoIteration1() {
```

## 为导入区域添加一个框架类

这是导入需要的泪，添加框架功能到这个复用：

```
■ // added iteration #1 添加 复用#1
■ import com.efg.framework.BlitSprite;
■ import com.efg.framework.TileByTileBlitSprite;
■ // end added iteration #1 结束添加 复用#1
```



## 定义复用 1 的变量

给已存在的 GameDemo.as 文件的变量定于区添加这些变量：

```

■ //movement specific variables added in iteration #1 详细移动变量添加到 复用#1
■     public static const MOVE_UP:int = 0;
■     public static const MOVE_DOWN:int = 1;
■     public static const MOVE_LEFT:int = 2;
■     public static const MOVE_RIGHT:int = 3;
■     public static const MOVE_STOP:int = 4;
■     /*** added iteration #1      添加      复用#1
■     //player specific variables      玩家详细  变量
■     private var player:TileByTileBlitSprite;
■     private var playerStartRow:int;
■     private var playerStartCol:int;
■     private var playerStarted:Boolean = false;
■     private var playerInvincible:Boolean = true;
■     private var playerInvincibleCountDown:Boolean = true;
■     private var playerInvincibleWait:int = 100;
■     private var playerInvincibleCount:int = 0;
■     /*** end added iteration #1 结束添加  复用#1

```

这些中的部分变量暂时还用不到，像控制角色无敌的这个，但是很快就会加入到游戏中。

接着，要往构造器中添加一些代码，并创建一些新的用来控制和 Main.as&框架类交互的函数，等会用到。你会注意到，我们把所有的东西都从构造器中移动到了一个叫做 init 的函数中，并在构造器中添加一行调用 init 的代码。为此，整段代码都列出来了。你应该删除之前的构造器函数并把下面的代码写在原位：

```

■     public function GameDemoIteration1() {
■         init();
■     }
■     private function init():void {
■         initTileSheetData();
■         player = new TileByTileBlitSprite(tileSheet, playerFrames, 0);
■         readBackGroundData();
■         drawLevelBackGround()
■         readSpriteData();
■         addChild(canvasBitmap);
■         newGame();
■     }
■     private function newGame():void {
■         newLevel();

```



```
■    }  
■    private function newLevel():void {  
■        restartPlayer();  
■    addChild(player);  
■    }  
■    private function restartPlayer(afterDeath:Boolean=false):void {  
■        trace("restart player");  
■        player.visible = true;  
■        player.currCol = playerStartCol;  
■        player.currRow = playerStartRow;  
■        player.x=(playerStartCol * tileWidth)+(.5*tileWidth);  
■        player.y = (playerStartRow * tileHeight) + (.5 * tileHeight);  
■        player.nextX = player.x;  
■        player.nextY = player.y;  
■        player.currentDirection = MOVE_UP;  
■        playerStarted = true;  
■        playerInvincible = true;  
■        playerInvincibleCountDown = true;  
■        playerInvincibleCount = 0;  
■    }
```

你注意到我们开始用一些会随后和框架类交互的 stub 函数模拟一个实际上的游戏循环（新游戏，新关卡，重新开始，等等）。既然还没有要处理的框架，就简单的按顺序调用类—init 调用 newGame ,newGame 调用 newLevel。

## 为 复用 1 的 init 函数

这个新的 init 函数主要是复制了之前的构造器函数，尽管添加了一行：

```
■ player = new TileByTileBlitSprite(tileSheet, playerFrames, 0);
```

这一行需要写在这里。把需要动画的玩家框架放在框架列表中，玩家才能创建实例化。如果你回到第六章，这些被配置到了 initTileSheet 函数中了。同样，readSpriteData 函数在玩家被创建前是不能执行的。如果玩家视觉元件没有在 readSpriteData 函数执行之前被创建的话，就会出现运行时错误，因为游戏引擎需要把玩家放在场景中，当然在创建之前是没法放的。

## 复用 1 的 restartPlayer（重新开始玩家）方法

这个方法/函数用来在新关卡或死后重新开始玩家。我们还不需要本地的 afterDeath 变量，但是过一会就需要了，因为当玩家确实死掉了，新的坦克替代死掉的那个，会拥有最大值的 healthPoints 和一些初始的弹药。当玩家活着进入到下一关时，这些属性是无效的。所以，传入 afterDeath 布尔值，让这些事件都能使用相同的函数方法。

这六行在设置玩家重新开始方面非常重要：



```

■ player.currCol = playerStartCol;
■ player.currRow = playerStartRow;
■ player.x=(playerStartCol * tileWidth)+(.5*tileWidth);
■ player.y = (playerStartRow * tileHeight) + (.5 * tileHeight);
■ player.nextX = player.x;
■ player.nextY = player.y;

```

我们这里做的第一件事是计算玩家刚进来的游戏网格上的当前行和列。我们要在关卡刚开始（或死后刚复活时）做这些工作，因为我们知道玩家这时是处在当前块的中心点的。

接着，我们要给玩家设置 x, y 轴，就用 player.currCol 和 player.currRow 变量。这里，我们找到玩家要进入的左上角的块的 x 值，位置是 playerStartCol\*tileWidth。我们给左上角的 y 值做一个相似的操作。尽管玩家坦克不会精确地坐在这个位置上。注意，玩家的位置会有额外（x 轴是 tileWidth\*0.5, y 轴是 0.5\*tileHeight）的变化。需要这个偏移，是因为 BlitSprite 里的 Bitmap 表明了玩家移动了 Sprite 注册点中心的宽的-0.5 倍和高的-0.5 倍。如果我们不把他重新定位到右边和下方多一点，它就会在左下方偏移 16 像素。

nextX 和 nextY 变量为玩家被设置成和当前 x y 值相等。这些很重要，因为它们要被用来更新游戏循环的一部分。我们会更新 nextX 和 nextY 值，然后基于它们（就像网格基于逻辑运动的评测）做碰撞测试，最后给游戏循环的渲染部分设置 x 和 y 相等等于 nextX 和 nextY。我们会通过游戏代码在随后的复用中仔细检查这些。

## readSpriteData 函数

readSpriteData 函数仅是 GameDemo.as 中的一个空成员。我们现在要用所需的代码填充它来循环 sprite 数据和把玩家放到场景上去：

```

■ private function readSpriteData():void {
■     var tileNum:int;
■     var spriteMap:Array = levelData.spriteMap;
■     for (var rowCtr:int = 0; rowCtr < mapRowCount; rowCtr++) {
■         for (var colCtr:int = 0; colCtr < mapColumnCount; colCtr++) {
■             tileNum = spriteMap[rowCtr][colCtr];
■             switch(tileSheetData[tileNum]) {
■                 case SPRITE_PLAYER:
■                     player.animationLoop = false;
■                     playerStartRow= rowCtr;
■                     playerStartCol= colCtr;
■                     player.currRow = rowCtr;
■                     player.currCol = colCtr;
■                     player.currentDirection = MOVE_STOP;
■                     break;
■             }
■         }
■     }

```



```
■      }  
■      }  
■      }
```

这个函数用来剖析 levelData.spriteMap 变量中的 2D 数组数据。当你调用时，这个变量被分配了原始的 GameDemo.as 里的 readBackGroundData 函数。这里慢镜头展示了这个函数里究竟发生了什么。

我们会用嵌套循环结构循环这个数据。嵌套循环结构包含一个能在 sprite 数据中复用行的外循环，在每一行中，我们会复用每一列。这个给每一行通过内部循环每一列的就是嵌套循环。

- 我们创建一个本地变量叫做 tileNum。当我们通过级别 sprite 数据复用时，这个会在我们的 sprite 层上给[row][column]储存当前块的 id。
- 我们把 levelData.spriteMap 的内容配置给 spriteMap 类变量的值。
- 当外循环小于 mapRowCount 数值 ( 15 ) 时，我们用本地的 rowCtr 变量 ( 初始值是 0 ) 开始我们的外循环。
- 当它小于 mapColumnCount ( 20)时，以用 colCtr 变量为 0 开始，内循环会循环当前行的每一列。
- 当前的 tileNum 是数组[rowCtr][colCtr]中的值。
- tileNum 被插入到 tileSheetData 单维数组中，来寻找处在那个位置的类型的 sprite。如果你调用，initTileSheetData 函数会剖析 TileSheetDataXML.as 文件数据，同样给 tileSheetData 数组的每一个 sprite 配置一个常数。
- 我们根据这个常数初始一个 switch:case 声明。
- 玩家变量被配置了他们的初始值。

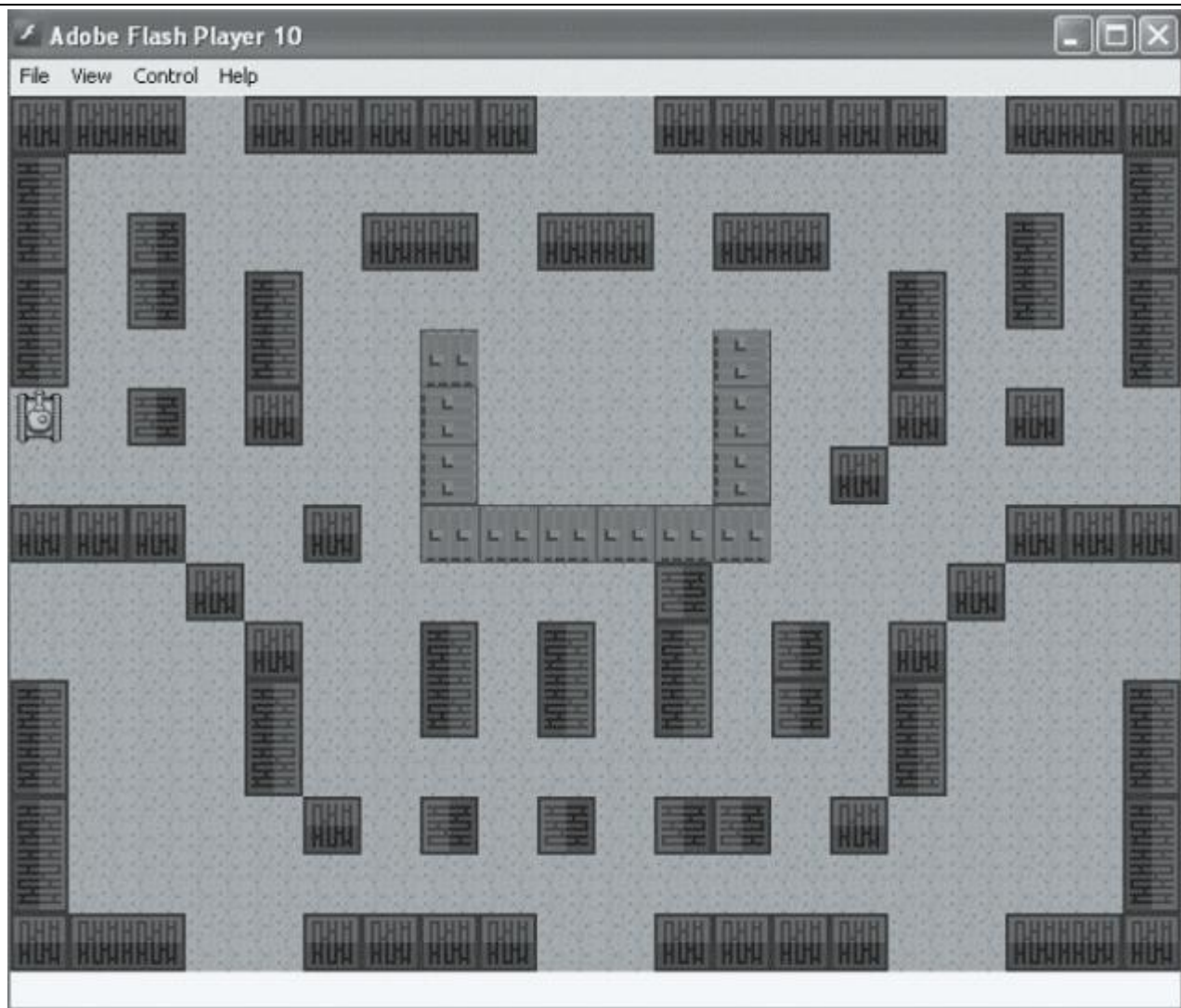
开始时 animationLoop 变量为 false。关卡开始时坦克师停止的 ( player.currDir=MOVE\_STOP )。我们不喜欢坦克在停止的时候步行动画，所以设置为了 false。

我们同样设置了 player.startRow 和 player.startCol。这些是用来在关卡开始时和玩家被摧毁建立新坦克到场景时放置玩家的。

Player.currCol 和 player.currRow 表示了当前角色的所在块，并且会在下面的复用中在许多游戏计算上使用。

## 测试复用 1

用你的开发环境，测试当前 GameDemoIteration1.as 文件的 迭代器 ( 之前翻译为 复用 )。这需要被设置为你的 FLA 或项目的文档或时时编译文件。如果你运行这个版本，你会看到和图 7-1 差不多的游戏场景。你的关卡或玩家坦克位置可能会根据你的创造有所不同。



图示 7-1：复用 1 的游戏画面

## 用关键逻辑来移动玩家（迭代器 2）

我们现在来解决玩家的移动，还通过隧道，如果对面的块是开放的他们会向另一边卷曲。（这句糊里糊涂的）在我们还没有把这个游戏加到框架前，我们必须解决的第一件事是给我们的游戏循环设置一个简单的 ENTER\_FRAME 事件。这个在与框架整合的时候会被替换。

我们会把这个文件的名称改为 GameDemoIteration2.as。你不必这么做，除非你想要保存每个迭代器的纪念品。如果你真的创建了一个新的文件，要确保这个类名称和构造器要改变的更适合这个新名称。如果你在使用 IDE，你需要确认这个文档类属性关联了新的类名。



这是 Flash IDE 的文件路径：

```
■ /source/projects/notanks/flashIDE/com/efg/games/notanks/GameDemoIteration2.as
```

这是 Flex SDK ( 用 Flash Develop ) 的路径

```
■ /source/projects/notanks/flexSDK/src/com/efg/games/notanks/GameDemoIteration2.as
```

## 为迭代器 2 改变类名

我们为游戏演示起了一个新的文件名，所以，我们现在需要相匹配的改变类名。我们同样需要改变构造函数名字。首先，我们来改类名。

```
■ public class GameDemoIteration2 extends Sprite
■ {
```

Next, we will change the constructor.

```
■ public function GameDemoIteration2() {
```

## 添加一个游戏定时器示例占位符

为这个游戏定时器示例占位符，我们会用一个标准的 ENTER\_FRAME 事件。我们的游戏一已经被设置成了最终要用 Main.as 的游戏定时器，所以所有这些代码在与 Main 类整合的时候都将被覆盖。尽管这样，现在还是能被用来演示所有的游戏逻辑，除了开始新关卡和死后新坦克出现时。

首先，我们必须导入在导入区导入事件包的代码：

```
■ import flash.events.*;
```

我们的游戏要接受键盘输入。游戏需要 Flash Player 的焦点以保证键盘输入被正确接受。我们不希望游戏的 Sprite 在聚焦时显示 focusRect，所以要把它关掉。

在 init 函数里，加上这行：



```
■ this.focusRect = false;
```

接着，我们必须找一个地方添加临时事件侦听器。我们已经选择在 `restartPlayer` 函数中做这个。在这部分只有这个函数的一部分会被列举出来。下面的粗体代码需要被添加进来。椭圆表示了函数中所有的其他代码，所以我们不必浪费地方重写一遍；我们会在此章中继续使用这个格式。

```
■ private function restartPlayer(afterDeath:Boolean=false):void {  
■     player.visible = true;  
■     player.currCol = playerStartCol;  
■     playerInvincibleCountDown = true;  
■     playerInvincibleCount = 0;  
■     ...  
■     /this is as good a place as any for now  
■     addEventListener(Event.ENTER_FRAME, runGame, false,0,true);  
■ }
```

接着，我们来添加一个非常基础的新的游戏循环函数/方法。创建接下来的新方法：

```
■ public function runGame(e:Event):void {  
■     trace("run game");  
■ }
```

如果你测试这段代码，会看到如下的输出：

run game

run game

run game

run game

输出会继续重复。我们的 `runGame` 方法被称为反复重复，但我们还不需要让它做任何事情。让我们在示例区块中去掉玩家移动。



## 用方向键切换玩家移动状态

在玩家实际运动之前,我们要先设置所有的这些个必需的函数来控制状态的改变。玩家的运动状态在 BlitSprite 类的实例属性 `currentDirection` 中。`currentDirection` 变量可以被设为五种状态之一,都是 `GameDemoInteraction2.as` 中的: `MOVE_UP`,`MOVE_DOWN`,`MOVE_LEFT`,`MOVE_RIGHT` 和 `MOVE_STOP`。

`currentDirection` 变量实际上并不能表示坦克的旋转(保存在 `Sprite.rotation` 中)。为此,我们可以用 `STOP` 作为一个方向。`TileByTileSprite` 可以有一个为 `MOVE_STOP` 的 `currentDirection` 和为 0 (面向上方)的旋转。同样(例如),`TileByTileBlitSprite` 可以有一个为 90 (面向右)的旋转和一个为 `MOVE_RIGHT` 的 `currentDirection`。但是,一个 `TileByTileBlitSprite` 不能让旋转为 -90 (面向左)和除了 `MOVE_RIGHT` (例如)以外的任何 `currentDirection`。这两个属性, `currentDirection` 和旋转,结合起来使运动更方便。

简言之,除非 `currentDirection` 为 `MOVE_STOP`,`currentDirection` 和旋转必须相匹配。

我们现在将在游戏文件里添加一些方法和代码,为的是能够检查被请求的运动是否符合逻辑还要用来实际改变玩家的旋转。一旦你加入了以下代码,就会显示出我们仅仅在旋转玩家,但是设置了移动玩家的一些根基。

## 添加按键的逻辑关系

第一个代码的改变很简单。我们需要添加一个变量来保存按键的值。按键会被这些简单的上和下事件掌控。当一个键被按下时,布尔值真就被放到一个叫做 `keyPressList` (在以 ASCII 键值表示的数组索引位置上)的数组中,当按键抬起事件被触发,这个数组索引位置会设为假。所以,例如,当用户按下时,就会给按下事件吧键码数值 37 发送到事件侦听器。这个侦听器然后就会在 `keyPressList[37]` 的位置放置为真。



让我们检查一下改变的所需代码，来让他工作起来。

首先，在变量的定义区，给 keyPressList 添加一行：

```

■      //iteration#2 variables, moving the player around the maze
■      private var keyPressList:Array = [];

```

The only changes to the newGame function are the addition of KEY\_DOWN and KEY\_UP event listeners:

```

■      private function newGame():void {
■
■          stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownListener);
■          stage.addEventListener(KeyboardEvent.KEY_UP, keyUpListener);
■
■          newLevel();
■      }

```

These two completely new listener functions must be added to the file:

```

■      private function keyDownListener(e:KeyboardEvent):void {
■
■          trace(e.keyCode);
■
■          keyPressList[e.keyCode]=true;
■      }
■      private function keyUpListener(e:KeyboardEvent):void {
■          keyPressList[e.keyCode]=false;
■      }

```

## 测试迭代器 2

你会希望注释掉 runGame 方法中的 `trace( "run game" );`声明，以便观察输出的关键事件。如果你

运行代码并按下每一个方向键，你会看到输出如下：

38

40

39



38, 40, 39 和 37 表示按下了每一个方向键。

## 给玩家运动更新运动状态（迭代器 3）

现在，我们去添加相当多的新方法掌控实际的运动逻辑。首先，让我们看看将被用来控制当前玩家运动状态的常量。我们打算给迭代器 3 改一个心的类名。

通常，我们会给你提供 Flash IDE 的文件路径：

/source/projects/notanks/flashIDE/com/efg/games/notanks/GameDemoIteration3.as

还有 Flex SDK(Flash Develop 开发)的路径：

/source/projects/notanks/flexSDK/src/com/efg/games/notanks/GameDemoIteration3.as

## 给迭代器 3 更改类名

我们给游戏演示起了个新文件名，所以，我们现在要改变类名来匹配。我们将同样需要改变构造器函数的名字。在此之前，我们先来改类名。

```
■ public class GameDemoIteration3 extends Sprite
■ {
■     Next, we modify the constructor name.
■     public function GameDemoIteration3() {
```

## 添加运动状态常量

下面的状态已经添加到了迭代器 1 的变量定义区，但是我们没有太多的讨论他们：

```
■ //movement specific variables
■ public static const MOVE_UP:int = 0;
```



```
■ public static const MOVE_DOWN:int = 1;
■ public static const MOVE_LEFT:int = 2;
■ public static const MOVE_RIGHT:int = 3;
■ public static const MOVE_STOP:int = 4;
```

当我们要给游戏角色计算当前方向和下一个方向时将会用这些声明。我们把他们声明为常量，是为了让他们比字符串或更容易记忆纯整数更容易记忆。另外，如果你在使用一个高级的编辑器（像 Flash Develop, Flex Builder, Flash Builder, 或 CS5 IDE），使用常量时会出现代码提示，很方便。

## 给迭代器 3 改变 runGame 方法

现在，我们给 runGame 方法添加一些代码来给每帧不同的运动方向检查 keyPressList 数组：

```
■ public function runGame(e:Event):void {
■     if (playerStarted) {
■         checkInput();
■     }
■ }
```

如你所见，我们只想在 playerStarted 被设置为真时运行 checkInput 方法。这样，在关卡改变和其他当你不想让玩家在场景中运动时，就可以不去给玩家计算按键了。

## 更新 CheckInput 方法

checkInput 方法是设置，计算和改变玩家的旋转和 currentDirection 属性的代码的核心。下面是代码展示。我们会在你查看（最好是自己动手输入）所有这些代码后指出主要概念。注意有些代码被注释掉了；你也要把这些行添加进去但是现在先保留他们的这种状态。这些代码等会会被用来执行播放声音和发射导弹。一些 trace 状态也被注释掉了。他们对于调试错误（debugging）比较有帮助。我们会在进展迭代器的过程中去掉它们的注释状态。



```
private function checkInput():void {
    //var playSound:Boolean = false;
    var lastDirection:int = player.currentDirection;

    if (keyPressList[38] && player.inTunnel==false) {
        if (checkTile(MOVE_UP, player )) {
            if (player.currentDirection == MOVE_UP || player.currentDirection ==
MOVE_DOWN || player.currentDirection == MOVE_STOP ) {

                switchMovement(MOVE_UP, player );
                //playSound = true;

            }else if (checkCenterTile( player)) {
                switchMovement(MOVE_UP, player );
                //playSound = true;
            }
        }else{
            //trace("can't move up");
        }
    }

    if (keyPressList[40] && player.inTunnel == false) {
        if (checkTile(MOVE_DOWN, player )) {
            if (player.currentDirection == MOVE_DOWN || player.currentDirection==
MOVE_UP || player.currentDirection == MOVE_STOP) {

                switchMovement(MOVE_DOWN, player );
                //playSound = true;
            }else if (checkCenterTile(player)) {

                switchMovement(MOVE_DOWN, player );
                //playSound = true;

            }
        }else {
            //trace("can't move down");
        }
    }
}
```



```
■      if (keyPressList[39] && player.inTunnel==false) {
■          if (checkTile(MOVE_RIGHT, player)) {
■              if (player.currentDirection == MOVE_RIGHT || player.currentDirection ==
MOVE_LEFT || player.currentDirection == MOVE_STOP) {
■
■                  switchMovement(MOVE_RIGHT, player );
■                  //playSound = true;
■
■              }else if (checkCenterTile( player)) {
■
■                  switchMovement(MOVE_RIGHT, player );
■                  //playSound = true;
■
■              }
■          }else {
■              //trace("can't move right");
■          }
■      }

■      if (keyPressList[37] && player.inTunnel==false) {
■          if (checkTile(MOVE_LEFT,player)) {
■              if (player.currentDirection == MOVE_LEFT || player.currentDirection ==
MOVE_RIGHT || player.currentDirection == MOVE_STOP) {
■
■                  switchMovement(MOVE_LEFT, player );
■                  //playSound = true;
■
■              }else if (checkCenterTile(player)) {
■
■                  switchMovement(MOVE_LEFT, player );
■                  //playSound = true;
■
■              }
■          }else {
■              //trace("can't move left");
■          }
■      }

■      if (keyPressList[32]&& player.inTunnel==false) {
■          //if (ammo >0) firePlayerMissile();
```



```
■      }  
■  
■      //if (lastDirection == MOVE_STOP && playSound) {  
■      //dispatchEvent(new  
      CustomEventSound(CustomEventSound.PLAY_SOUND,Main.SOUND_PLAYER_MOVE, false,  
      1, 0));  
■      //}  
■      }
```

基于块形式的碰撞检测有很多方法。我们在这里给这个游戏理性执行一组特别的逻辑。这不是（例如）很适合那种超自由运动的驾驶类游戏。幸运的是，我们会在第十章讨论那些。

这些是你需要知道的关于 checkInput 方法的几个要点：

- 如果玩家正在隧道中前进，游戏是不接受玩家的键盘输入的。这极大地简化了玩家不在场景中（在隧道中）时控制运动的代码。TileByTileBlitSprite 类的 inTunnel 属性要在这个评估中使用。当玩家离开场景并进到隧道里，这个会被设置为真。
- checkTile 方法是用来检查玩家试图进入的下一个区块是否有效 TILE\_MOVE 块（参见第六章，观察 TILE\_WALL 和 TILE\_MOVE 的不同）。
- 如果 checkTile 方法对于正准备前进的方向返回真，就会有更多的检测。这些检测大多取决于玩家是否打算转弯。转弯在这种类型的运动结构中最复杂的步骤，因为角色只能转到一个有道路（不是墙）的方向。
- 如果要移动的方向和当前方向相同或相反，或者角色是停止的，运动是则有效的并且会调用 switchMovement 方法。
- 如果要移动的方向不是当前或相反方向，角色也不是停止的，我们必须进行更多的计算。这意味着玩家正准备转弯。角色只能在当前块的中心点才被允许转弯。这种情况由调用 checkCenterTile 掌控。



我编写的这个游戏中的运动模仿了一些吃豆人的属性。在吃豆人中，玩家正能在下一个块为 TILE\_WALK 块时才能改变方向。一些人可能会想这意味着玩家坦克如果被卡在角落里的时候就不能恢复自由了。这个假设应该是错误的。就像吃豆人，坦克是不会被卡在任何角落里的，因为玩家随时可以按方向键向相反的方向运动而不必转到哪里。

## 添加 checkTile 方法

checkTile 方法只有一个目的：计算 TileByTileBlitSprite 的实例是否能合法的移动进一个块里。这能做这些，是用了这种方法，首先，计算下一个块会是什么，然后在 tileSheetData 数组中检查这个快的数值，以查看它是否是一个 TILE\_MOVE 或 TILE\_WALL 块。这个方法接受 direction 中用整数值表示的方向，还有用对象变量表示的 TileByTileByTileSprite 实例。

```
■ private function checkTile(direction:int, object:TileByTileBlitSprite):Boolean {
■
■     var row:int = object.currRow;
■     var col:int = object.currCol;
■
■     switch(direction) {
■         case MOVE_UP:
■             row--;
■             if (row < 0) {
■                 row = mapRowCount-1;
■             }
■
■             break;
■
■
■         case MOVE_DOWN:
■             row++;
■             if (row == mapRowCount) {
■                 row = 0;
■             }
■             break;
```



```
■  
■  
■      case MOVE_LEFT:  
■          col--;  
■          if (col < 0) {  
■              col = mapColumnCount - 1;  
■          }  
■          break;  
■  
■      case MOVE_RIGHT:  
■          col++;  
■          if (col == mapColumnCount) {  
■              col = 0;  
■          }  
■          break;  
■  
■      }  
■  
■  
■  
■      if (tileSheetData[levelTileMap[row][col]] == TILE_MOVE) {  
■          //trace("can move");  
■          return true;  
■      }else {  
■          //trace("can't move");  
■          return false;  
■      }  
■  
■      }  
■  
■      }
```

这里是几个关于 checkTile 的要点：

- 当传入方向时，checkTile 方法会运行一个 switch : case 语句。
- 它要做的第一件事是设置自身的当地 row 和 col 变量相等于对象( 这里是玩家 )的 currRow 和 currCol 变量。
- 根据玩家试图前进的方向，col 和 row 的值也会根据下面的一个来增加或减少：

Left : col--

Right : col++

Up : row--



Down : row++

- 如果行或列的新值不在场景中发生( 可能是隧道里 ),则 col 或 row 块值会被设置成游戏场景的对立面。
- 在 switch:case 声明之后 ,tileSheetData 数组也被计算完毕了。它会把 row 和 col 值插进 levelTileMap 的关卡块的 2D 数组里 ,以取得一个数值来表示关卡地图上区块 ( tileSheetData ) 中的块。如果那个块是一个 TILE\_MOVE 块 ,函数就会转为真。如果不是 ,就会为假。

## 添加 checkCenterTile 方法

checkCenterTile 方法简单的带着传入的 TileByTileBlitSprite 实例 ,还有计算它是不是在当前块的中心。如果是 ,它返回真 ;如不是 ,返回假。

```
■ private function checkCenterTile( object:TileByTileBlitSprite):Boolean {  
■     var xCenter:int = (object.currCol * tileWidth) + (.5 * tileWidth);  
■     var yCenter:int= (object.currRow * tileHeight) + (.5 * tileHeight);  
■  
■     if (object.x == xCenter && object.y == yCenter) {  
■         //trace("in center tile");  
■         return true;  
■  
■     }else {  
■  
■         //trace("not in center tile");  
■         return false;  
■     }  
■ }
```

块的 x 和 y 值表示的是块的左上角的位置。要想知道当前 x 和 y 的值 ,需要用 currCol 乘以 tileWidth 和 currRow 乘以 tileHeight。然后 ,给 x 值加上 tileWidth 的一半 ,给 y 值加上 tileHeight 的一半。这是来计算等会要给我们的对象检查 x 和 y 值的 xCenter 和 yCenter 值的。游戏对象从块的中心偏移了一些 ,因为它们处于 Sprite 持有者的中点。如果他们不是这样 ,我们也将必须添加 tileWidht 和 tileHeight 的一半到对象的 x 和 y 值上。



## 添加 switchMovement 函数

switchMovement 函数把 direction 变量作为参数来表示对象将要移动的方向。它把一个 TileByTileBlitSprite 参引作为对象参数变量。基于被传入的方向这个函数简单的设置这个对象的属性匹配到新的方向。这里是函数的代码：

```
■ private function switchMovement(direction:int, object:TileByTileBlitSprite):void{
■     switch(direction) {
■         case MOVE_UP:
■             //trace("move up");
■             object.rotation = 0;
■             object.dx = 0;
■             object.dy = -1;
■             object.currentDirection = MOVE_UP;
■             object.animationLoop = true;
■             break;
■
■         case MOVE_DOWN:
■             //trace("move down");
■             object.rotation = 180;
■             object.dx = 0;
■             object.dy = 1;
■             object.currentDirection = MOVE_DOWN;
■             object.animationLoop = true;
■             break;
■
■         case MOVE_LEFT:
■             //trace("move left");
■             object.currentDirection = MOVE_LEFT;
■             object.rotation = -90;
■             object.dx = -1;
■             object.dy = 0;
■             object.animationLoop = true;
■             break;
■
■         case MOVE_RIGHT:
■             //trace("move right");
```



```
■      object.currentDirection = MOVE_RIGHT;
■      object.rotation = 90;
■      object.dx = 1;
■      object.dy = 0;
■      object.animationLoop = true;
■      break;
■
■      case MOVE_STOP:
■          //trace("move stop");
■          object.currentDirection = MOVE_STOP;
■          object.dx = 0;
■          object.dy = 0;
■          object.animationLoop = false;
■          break;
■      }
■
■      object.nextRow = object.currRow + object.dy;
■      object.nextCol = object.currCol + object.dx;
■  }
■
```

这些是你需要知道的关于 switchMovement 函数的一些要点：

- 根据传入的方向值,它将设置被传入的 TileByTileBlitSprite 对象属性的 currentDirection 匹配到持久移动的状态变量。
- 根据方向变量,它设置对象的旋转以匹配。
- 根据方向变量,这个对象的 dx 和 dy 属性会被改变。这些是在更新函数中使用来沿着游戏区域的水平和垂直方向移动的。因为我们只移动上下左右,其中一个总是 1 (或-1),而其余的总是 0,除非对象停止了,在这种情况下,他们都是 0。
- 如果物体在一个方向上移动 (没有停止),它的 object.animationLoop 属性会设置为 true。
- 如果物体停止了, object.animationLoop 属性会被设置为 false。
- 最后,这个对象的 nextRow 和 nextCol 属性会被设置。

## 测试迭代器 3

当你运行这个迭代器，你会发现可以旋转坦克到任何方向，除了紧挨着的下一个是 TILE\_WALL 块的方向。在图 7-2 中，你发现坦克指向下方，它不能指向上方，但是他可以指向左和右。如果这看起来有限制性也不用担心。当我们添加坦克的运动到下一个迭代器中，你会发现这种运动非常适合这种类型的游戏。

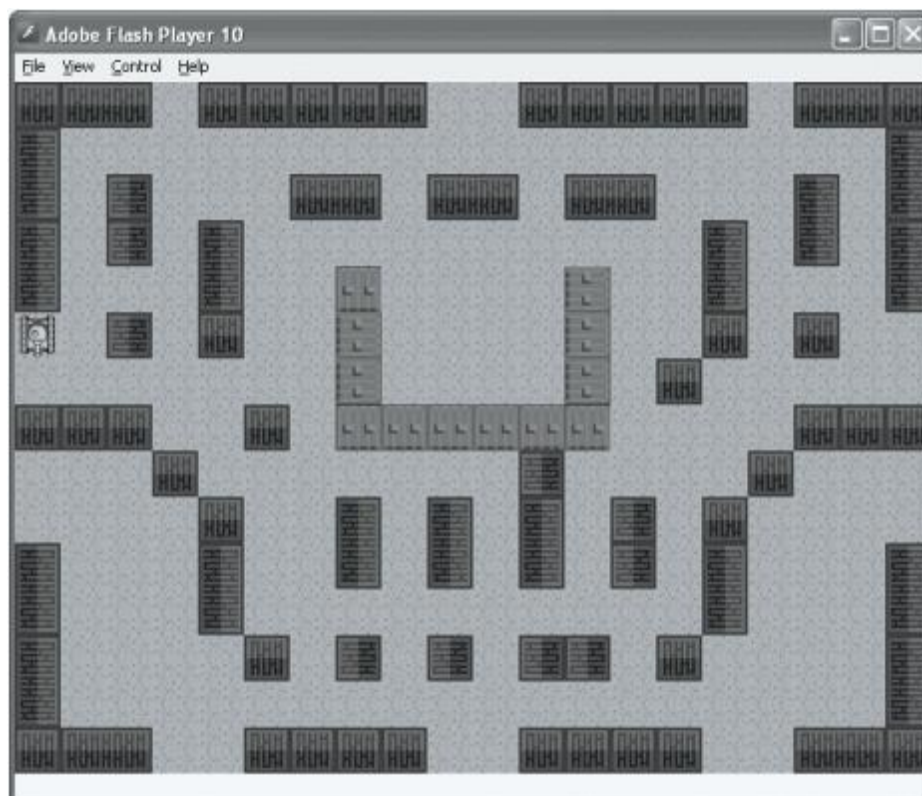


图 7-2 迭代器 3 游戏场景

## 更新并渲染玩家的移动（迭代器 4）

接下来，我们要给游戏代码添加更新和渲染的函数。当我们完成这些后，玩家坦克就可以在地图中平滑的巡视了，从这个块中心移动到下一个块中心，像一个专家一样在隧道间穿梭。

在这个部分，我们要给我们的游戏创建更新和渲染部分，还包括只有玩家需要的代码。我们同样会更新变量定义区，给更新和渲染函数的新的调用添加 runGame 函数。

开始吧，我们要给这个迭代器更改一个新的类名。



这是我们 Flash IDE 用的心的类路径：

```
/source/projects/notanks/flashIDE/com/efg/games/notanks/GameDemoIteration4.as
```

这是 Flex SDK 的路径：

```
/source/projects/notanks/flexSDK/src/com/efg/games/notanks/GameDemoIteration4.as
```

## 给迭代器 4 更改类名

我们给这个游戏演示版起了个新名字，所以，我们现在需要改类名以匹配。我们同样需要改变构造器函数的名字。首先，我们要改变类名。

```
1 . public class GameDemoIteration4 extends Sprite
2 . {
3 . And as before, we next change the constructor.
4 .     public function GameDemoIteration4() {
```

## 为迭代器 4 更新变量定义

我们需要添加一些变量来标明我们游戏场景的四个边界。这么做是有必要的，因为所有的游戏角色的位置都有 16 像素的块宽和块高的偏移。这些偏移被加上来弥补 BlitSprite 里的 Bitmap 被放到了 x 轴的  $-5 * \text{tileWidth}$  和 y 轴的  $-5 * \text{tileWidth}$  上（要把它放到注册点的中心）。我们同样设置一些变量来为角色储存在隧道里的最大和最小位置（ $x_{\text{Max}}, y_{\text{Max}}, x_{\text{Min}}$  和  $y_{\text{Min}}$ ）。例如，实际的水平最小值是 16（不是 0），因为玩家和敌坦克都在 BlitSprite 上偏移了 -16 像素，以便让他们在中心点旋转。如果我们用 0 作为最小水平运动，坦克的一半实际上可能就会偏出了场景——如果坦克在  $x=0$  的左上角的话。最大水平移动位置是 624（ $624 + 16 = 640$ ），原因是一样的。垂直方向上的最大和最小移动位置很相似。

隧道移动位置标明在它出现在场景对侧面之前场景移动物体偏移多远。

把下面的代码添加到 GameDemoIteration4.as 文件的变量定义区：

```
1 . /** added iteration #4
2 . private var xMin:int = 16;
3 . private var yMin:int = 16;
```



```
4 . private var xMax:int = 624;
5 . private var yMax:int = 464;
6 . private var xTunnelMin:int = -16;
7 . private var yTunnelMin:int = -16;
8 . private var xTunnelMax:int = 656;
9 . private var yTunnelMax:int = 496;
10 . /** end iteration #4
```

你一会就会看到这些数字是如何被用来限制游戏对象的最大和最小运动的，还允许隧道包裹效果在关卡中的任何一侧看起来都一样。

## 给迭代器 4 更新 runGame 函数

我们要给 runGame 函数添加两个简单的调用函数，以便快速调用更新和渲染函数。整个的新的 runGame 函数应该是这个样子：

```
1 .      public function runGame(e:Event):void {
2 .          if (playerStarted) {
3 .              checkInput();
4 .          }
5 .          /*** added iteration #4
6 .          update();
7 .          render();
8 .          /** end added iteration #4
9 .      }
```

## 添加更新函数

更新函数的首要任务就是更新玩家（和其他的运动物体）的 nextX 和 nextY 属性。每次更新，这两个属性就会最终被进行碰撞检测（直到下稍后的迭代器之前是不会完成的）。

```
1 .      private function update():void {
2 .          /*** added iteration #4
3 .          player.nextX = player.x + player.dx*player.moveSpeed;
4 .          player.nextY = player.y + player.dy * player.moveSpeed;
5 .
6 .          if (player.y <yMin || player.y>yMax || player.x<xMin || player.x >xMax){
7 .              player.inTunnel = true;
```



```
8 .      }else {
9 .          player.inTunnel = false;
10 .      }
11 .
12 .      if (player.inTunnel) {
13 .
14 .          switch(player.currentDirection) {
15 .
16 .              case MOVE_UP:
17 .                  if (player.nextY == yTunnelMin){
18 .                      player.nextY=yTunnelMax
19 .                  }else if (player.nextY == yMax) {
20 .                      player.inTunnel = false;
21 .                  }
22 .                  break;
23 .
24 .              case MOVE_DOWN:
25 .                  if (player.nextY == yTunnelMax){
26 .                      player.nextY = yTunnelMin;
27 .                  }else if (player.nextY == yMin) {
28 .                      player.inTunnel = false;
29 .                  }
30 .                  break;
31 .
32 .              case MOVE_LEFT:
33 .                  if (player.nextX == xTunnelMin){
34 .                      player.nextX = xTunnelMax;
35 .                  }else if (player.nextX == xMax) {
36 .
37 .                      player.inTunnel = false;
38 .                  }
39 .                  break;
40 .
41 .              case MOVE_RIGHT:
42 .                  if (player.nextX == xTunnelMax) {
43 .                      player.nextX = xTunnelMin;
44 .                  }else if (player.nextX == xMin) {
45 .                      player.inTunnel = false;
46 .                  }
47 .                  break;
```



```
48 .
49 .             case MOVE_STOP:
50 .                 trace("stopped");
51 .                 break;
52 .             }
53 .         }
54 .         player.currRow = player.nextY / tileWidth;
55 .         player.currCol = player.nextX / tileHeight;
56 .
57 .         player.updateCurrentTile();
58 .         /*** end added iteration #4
59 .     }
```

这里是你要知道的关于更新函数的几个要点：

- 玩家对象的 nextX 和 nextY 属性是通过添加与对象的 moveSpeed 相乘的 dx 和 dy 值来更新的。  
  
player.moveSpeed 值是 2。应该注意到，当我们试图检测块的中心时，块宽和高必须被任何物体的 moveSpeed 平均的分开。moveSpeed 值在 BlitSprite 类中被默认的设置 2，但是他可以被设置为 1, 2, 4, 8, 16 和 32。1, 2, 4 和 8 会得到最好的结果和最好的角色控制体验。千万不要用除了 1 以外的奇数，否则，游戏对象将可能永远碰不到块的中心。
- 如果你曾玩过吃豆人类型但是控制性不好的游戏，你会发现当你实际上想转弯时很容易穿插过隧道。  
  
我们这里执行的控制方案会竭尽全力的确保控制缺失的情况不会发生。当角色到达块的中心时，我们实际上停止了玩家的向前移动，除非一直摁着让角色向没路的方向移动。如果玩家按下一个方向键来转弯，当到达中心时转弯就会发生。

如果我们做了一个吃豆人游戏，这个里面的角色在到达每个块的中心点是不会停下来，我们只需排除 render 函数里调用的 switchMovement(MOVE\_STOP) 方法

- 如果玩家在隧道里，大部分更新的代码被起用，因为一旦玩家移开了屏幕，我们需要控制角色并把它



弄到另一边。在这个迭代器的最后，你会注意到一旦你进入隧道，你就不能转弯直到隧道的另一端之前。这个过程很快，玩家不会注意到，但是这么使用让我们能够查出何时玩家进入和离开隧道，还允许圆滑的动画进入隧道在另一端出来。

- 无论玩家是不是在隧道里，我们常常会基于 `nextY` 和 `nextX` 属性实时更新 `player.currRow` 和 `player.currCol`。
- 我们要做的最后一件事是调用 `player.updateCurrentTile` 函数，从讨论的 `BlitSprite` 类里重新调用这个？如果不是，再看一次。如果 `player.animationLoop` 布尔值为真，在到达 `animationDelay`（三帧）是代码就要更新 `currentTile` 属性，`doCopyPixels` 布尔值就会被设置为真。如果不是，它仍保持为假。

## 添加 render 函数

`render` 函数关系到三个非常重要的任务，最重要的就是调用玩家对象的 `renderCurrentTile` 函数。这

里是这个迭代器的 `render` 函数完整代码：

```
1 .      private function render():void {
2 .          player.x = player.nextX;
3 .          player.y = player.nextY;
4 .
5 .          if (checkCenterTile(player)) {
6 .
7 .              if (!player.inTunnel) {
8 .                  switchMovement(MOVE_STOP, player);
9 .                  //setCurrentRegion(player);
10 .              }
11 .          }
12 .          player.renderCurrentTile();
13 .
14 .      }
```

这里是关于 `render` 函数的比较重要的几点，你最好了解一下：

- 这个函数最后会在 `checkCollisions` 函数之后调用（还未就绪）。当与 `nextX` 和 `nextY` 的碰撞检测完之后，`BlitSprite` 物体的实际 `x` 和 `y` 就会被更新，并直接反射到场景中。
- `render` 函数的另一个任务是检测玩家是否在当前块的中心。这仅仅在 `32*32` 的块上当块的 `moveSpeed` 属性设置为 `1`，`2`，`4`，`8`，`16` 和 `32` 时有效。
- 如果坦克角色在当前块的中心但不在隧道中，它就会停止。乳沟角色不在当前块的中心，他就不会停



止。如果玩家按住一个前面没有路的方向键而且到了块的中心，停止看起来不会那么明显，玩家坦克会继续移动到下一帧。

- 我们要做的最后一件事是调用 `player.renderCurrentTile` 函数。在讨论的 `BlitSprite` 类中重新调用次函数？如果不是，再看一次。如果 `player.doCopyPixels` 布尔值是真，坦克轨迹会通过从 `tileSheet` 复制 `BlitSprite` 实例里相关的位图像素到 `bitmapData` 属性。
- 注意我们把 `setCurrentRegion` 函数注释掉了。我们很快就会用到它。

## 测试迭代器 4

当你运行这个迭代器，你会发现玩家坦克可以在场景中块之间平滑的移动，还可以随意进出左右边界的“时空”隧道（见图 7-3，给这个迭代器的场景截图）。注意，当你在它在块之间运动时按下运动键，坦克是不会停下来的。同样，如果坦克在向上移动，而你想让它移动到左边的有开口的“隧道”（比如），你必须把手拿开上方向键并按下左方向键。坦克就会在不停止的情况下连贯的转弯。

我们现在已经完成了移动游戏角色的大部分代码。而敌坦克的代码会是非常相似的。我们会简短的看一下，然后在下一个迭代器中进入到“足够好”的敌坦克的 AI 中去。

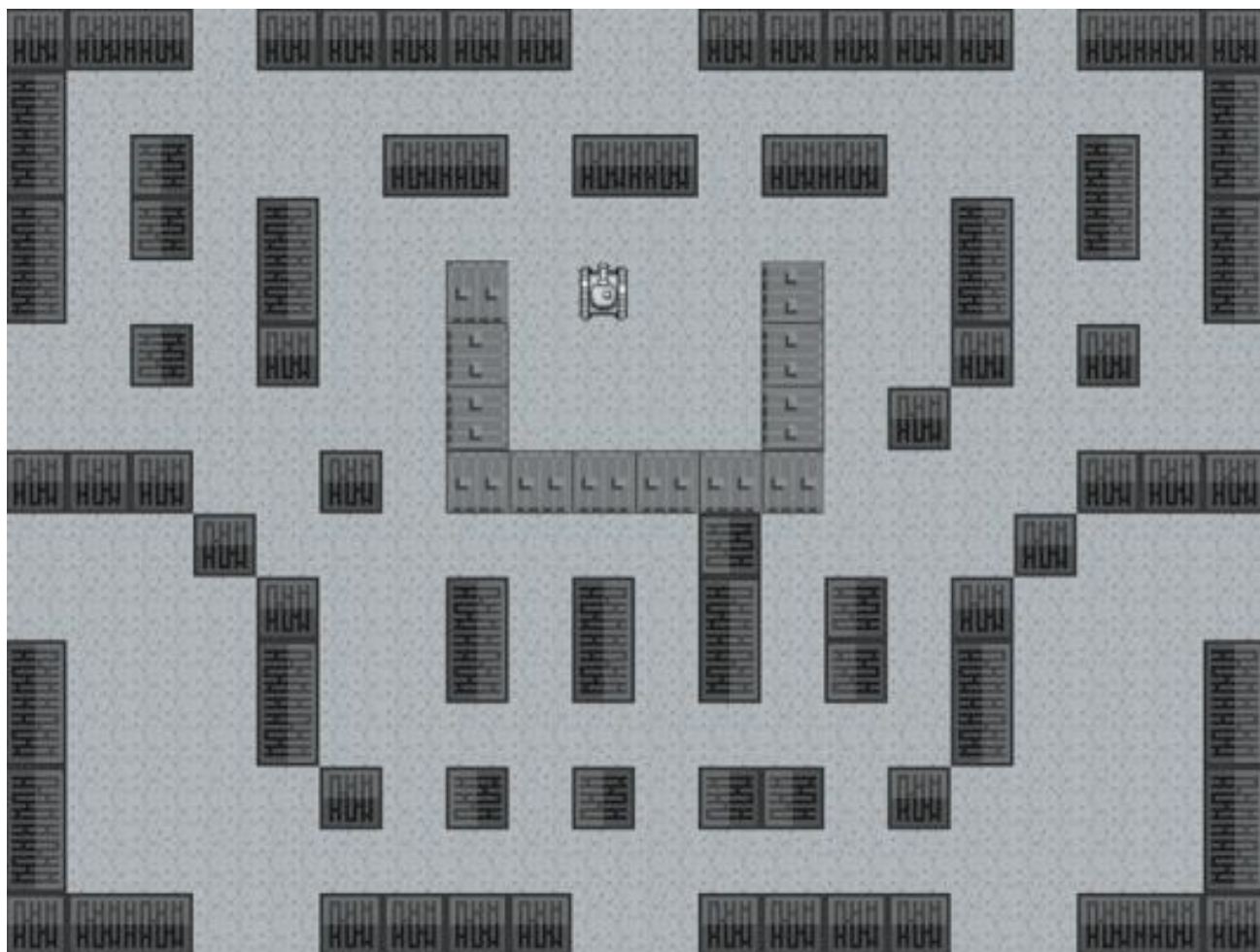


图 7-3 迭代器 4 的游戏场景

## 添加和移动敌坦克（迭代器 5）

敌坦克有一个职责——摧毁玩家坦克。我们会给这个坦克创建一个类似于玩家坦克行为方式的游戏逻辑。

一个不同之处在于敌坦克会有一些适当的智能 AI 代码，确保它到至少能让他们挑战一下人类的水平。

在这个迭代器中，我们会给游戏重新组织一下以更接近最终版。这意味着重新组织和移动一些当前的代码，并添加一些代码把坦克布置到场景上去。

我们还要为这个迭代器更改一个新的类名。



Flash IDE 的文件路径：

/source/projects/notanks/flashIDE/com/efg/games/notanks/GameDemoIteration5.as

SDK 的文件路径：

/source/projects/notanks/flexSDK/src/com/efg/games/notanks/GameDemoIteration5.as

## 给迭代器 5 更改类名

我们再一次为这个游戏演示有了一个新的文件名，所以那，现在就更改类名来匹配它。我们还需要更

改构造器函数的名字。先来改类名

```
■ public class GameDemoIteration5 extends Sprite
■ {
■     and then change the constructor name:
■     public function GameDemoIteration5() {
```

让我们去看看这个迭代器中的变量定义区的变动。

## 给迭代器 5 更新变量定义

我们需要在这个迭代器中给敌坦克添加一些变量以便成功的加到场景中。同样，我们要给游戏场景添加一些包含区域信息的变量。

一会在我们说到代码的敌人 AI 部分时会详细的讨论区域的问题，但现在，只要注意我们把场景分成了四份就行了。这些区域被粗略的划分，有 TOP\_LEFT, TOP RIGHT, BOTTOM LEFT 和 BOTTOM RIGHT。这些会被用来引发敌人的运动。如果玩家和敌人在相同的场景区域，那么敌人就开始追逐玩家（一会会在“追赶玩家”部分中详解）。

在变量定义区添加这些变量：



```
■    /** added in iteration #5
■    private var regions:Array;
■    private var tempRegion:Object;
■    private var enemyList:Array;
■    private var tempEnemy:TileByTileBlitSprite;
■
■    //end iteration #5
```

## 更新 init , newGame 和 newLevel 函数

为迭代器 5 所作的变更都是关于构造器和修改游戏代码的，所以在最终版本中融合了 Main 类和游戏框架

是比较好的。情完整的拷贝下面的这些函数：

```
■    private function newGame():void {
■
■        /** changed in iteration #5
■        setRegions();
■        stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownListener);
■        stage.addEventListener(KeyboardEvent.KEY_UP, keyUpListener);\
■        addChild(canvasBitmap);
■        newLevel();
■        /** end changed in iteration #5
■
■    }
■
■    private function init():void {
■        this.focusRect = true;
■        initTileSheetData();
■        newGame();
■    }
■
■    private function newLevel():void {
■        /** changed in iteration #5
■        enemyList = [];
■        player = new TileByTileBlitSprite(tileSheet, playerFrames, 0);
```



```
■ readBackGroundData();
■ readSpriteData();
■ drawLevelBackGround();
■ restartPlayer();
■ addChild(player);
■ /** end changed in iteration #5
■ }
```

值得注意的一个变更是 setRegions 函数调用 newGame 函数。我们等会测试这个函数。他为敌人 AI 追逐把游戏场景分成了四份逻辑区域。你看看这个函数中剩余的代码，我们只是把它重新排序来让它与框架更兼容。我们同样添加了一个新行，以重新设置 enemyList 数组到默认的空的[] 设置。我们会在每关开始时做这些，以确认我们有一个干净的，崭新的数组来给关卡贮藏敌人。

## 添加 setRegions 函数

我们已经把游戏场景分成了一下四块逻辑区域：

- 左上区开始于 列 0，行 0，结束语于列 9，行 7.
- 右上区开始于 列 10，行 0，结束于列 19，行 7.
- 左下区开始于 列 0，行 8，结束于 列 9，行 14.
- 右下区开始于 列 18，行 8，结束于 列 19，行 14.

详见图 7-4.

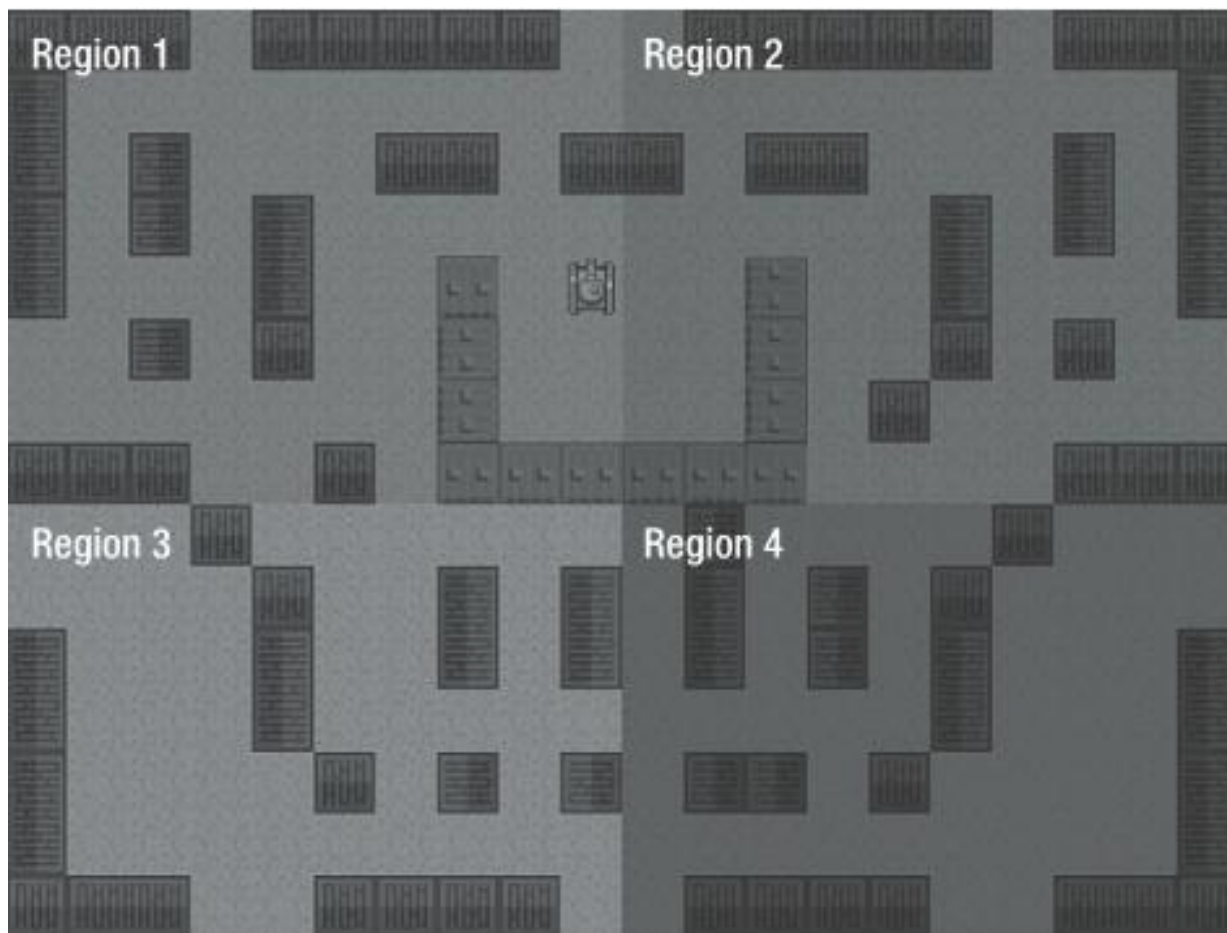


图 7-4 迭代器 5 的区域划分，以不同程度的阴影表示

如图 7-4 所示，既然在垂直方向上有 15 个块，我们必须把上面的区域划为 8 个，而下面的为 7 个的高度。可以与这种方式不同，但是这么做容易些。水平的长度（各 10 块）左右相同是因为我们有 20 各块。

现在，去看看这些区域需要的两个函数。第一个，`setRegions`，是用来创建和填充区域数组。第二个，`setCurrentRegion`，给一个对象计算当前区域。把这些输入到你的 `GameDemoIteration5.as` 文件里。

```
■ /** added in iteration #5  
■ private function setRegions():void {  
■ regions = [];  
■  
■ //top left region
```



```
■      var tempRegion:Object = { col1:0, row1:0, col2:9, row2:7 };
■      regions.push(tempRegion);
■
■      //top right region region
■      tempRegion = { col1:10, row1:0, col2:19, row2:7 };
■      regions.push(tempRegion);
■
■      //bottom left region region
■      tempRegion = { col1:0, row1:8, col2:9, row2:14 };
■      regions.push(tempRegion);
■
■      //bot right region region
■      tempRegion = { col1:10, row1:8, col2:19, row2:14 };
■
■      regions.push(tempRegion);
■  }
■
■
■
■  private function setCurrentRegion(object:TileByTileBlitSprite):void {
■
■      var regionLength:int = regions.length - 1;
■
■      for (var regionCtr:int = 0; regionCtr <= regionLength; regionCtr++) {
■          tempRegion = regions[regionCtr];
■          if (object.currCol >= tempRegion.col1 && #
■              object.currCol <= tempRegion.col2 && #
■              object.currRow >= tempRegion.row1 && #
■              object.currRow <= tempRegion.row2) {
■
■              object.currentRegion = regionCtr;
■          }
■      }
■  }
■  }/** end added in iteration #5
```

setCurrentRegion 函数包括了一个 TileByTileBlitSprite 就像一个参数调用一个对象。使用对象的 currCol 和 currRow 属性，他会在区域数组中循环这四个区域来寻找当前对象在哪一个区域。对象的 currentRegion 属性会变为检测到的区域数。



## 变更 readSpriteData 函数

我们现在需要修改 readSpriteData 函数来创建一个新的敌坦克并把它们添加到 enemyTanks 数组中。

在已存在 readSpriteData 函数中的的 switch : case 声明中，你需要给敌坦克添加一个情况。添加下面的代码。我们已经包含了 SPRITE\_PLAYER 的情况以防万一。这应该匹配了你已经在 GameDemoIteration5.as 文件中准备好的内容。

```
■          case SPRITE_PLAYER:
■              player.animationLoop = false;
■              playerStartRow= rowCtr;
■              playerStartCol= colCtr;
■              player.currRow = rowCtr;
■              player.currCol = colCtr;
■              player.currentDirection = MOVE_STOP;
■              break;
■
■          /** added in iteration #5
■
■          case SPRITE_ENEMY:
■
■              tempEnemy = new TileByTileBlitSprite(tileSheet, enemyFrames, 0);
■
■              tempEnemy.x=(colCtr * tileWidth)+(.5*tileWidth);
■              tempEnemy.y = (rowCtr * tileHeight) + (.5 * tileHeight);
■
■              tempEnemy.currRow = rowCtr;
■              tempEnemy.currCol = colCtr;
■
■              setCurrentRegion(tempEnemy);
■
■              tempEnemy.currentDirection = MOVE_STOP;
■              tempEnemy.animationLoop = false;
■
■              addChild(tempEnemy);
■              enemyList.push(tempEnemy);
■
■              break;
■          /** end added in iteration #5
```

## 测试迭代器 5

当你测试这个迭代器，你应该能看到所有的坦克被部署到场景中了，就像图 7-5 所示。

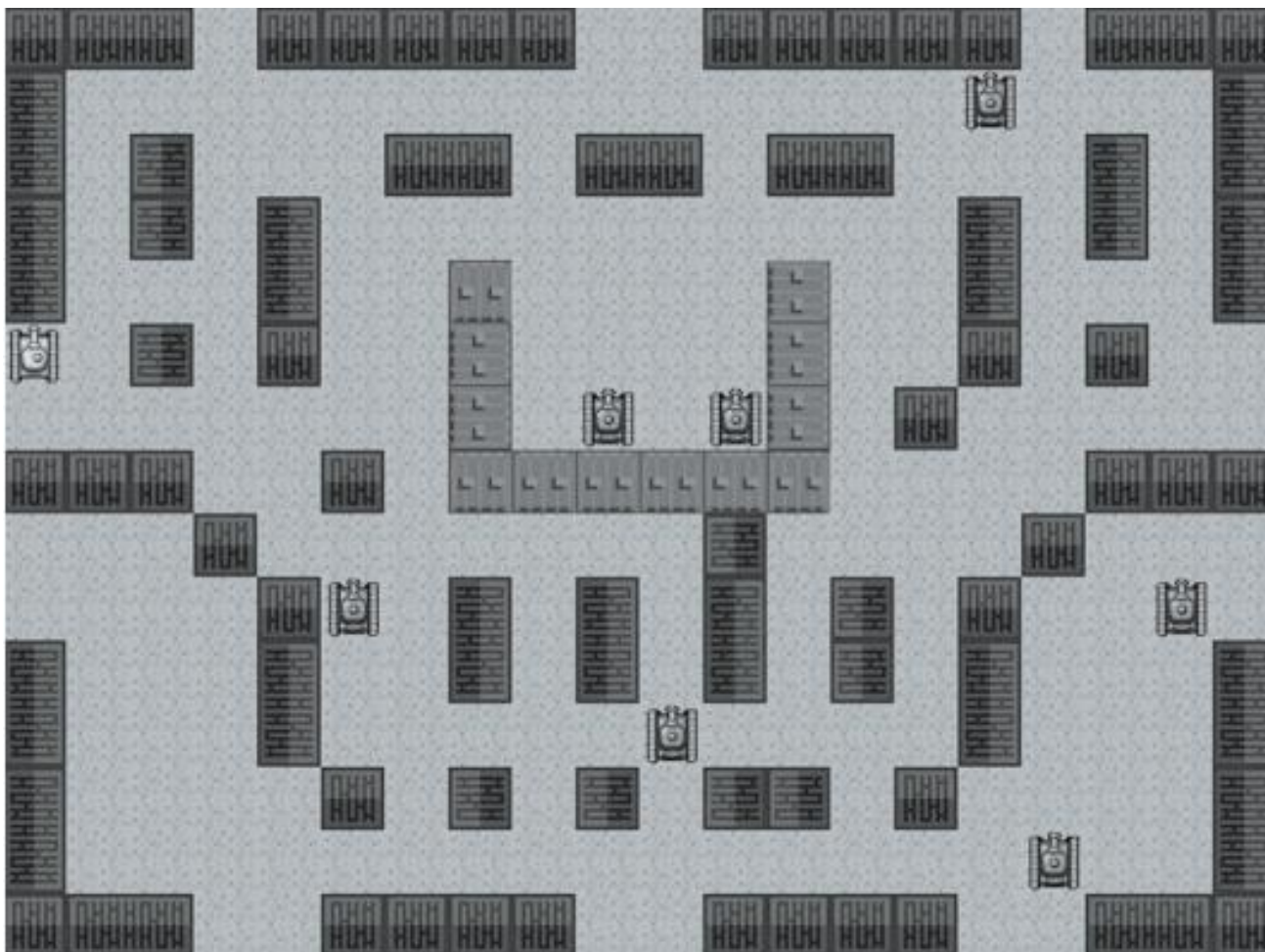


图 7-5 迭代器 5 的游戏场景和坦克

## 用 AI 使敌坦克运动（迭代器 6）

现在敌坦克已经在场景中了，我们必须让他们追赶玩家。我们要做的一件事是设置玩家自己的区域。

一旦完成这个，我们就可以添加 AI 和运动代码到敌坦克了，这样它们就可以开始追赶玩家了。

改类名。



Flash IDE 路径

/source/projects/notanks/flashIDE/com/efg/games/notanks/GameDemoIteration6.as

SDK

/source/projects/notanks/flexSDK/src/com/efg/games/notanks/GameDemoIteration6.as

## 给迭代器 6 改类名

这个游戏演示有了个新的文件名，所以，我们需要改类名来匹配它。我们同样需要改构造器函数的名字。首先，来改类名。

```
■ public class GameDemoIteration6 extends Sprite
■ {
■     Then, we modify the constructor.
■     public function GameDemoIteration6() {
```

## 给 restartPlayer 函数一些添加

首先，我们要给 restartPlayer 函数添加一行。你只需添加一面代码的粗体部分，其余的应该已经在你的代码中有了，写在这里仅为上下文参考：

```
■         playerInvincibleCount = 0;
■
■         /** added iteration 6
■         setCurrentRegion(player);
■         /** end added iteration 6
■
■         //this is as good a place as any for now
■         addEventListener(Event.ENTER_FRAME, runGame, false,0,true);
```

然后，去看看设置玩家的 currentRegion 属性的 render 函数的变化。我们需要的这行已经添加到 render 函数中了。在他前面有注释标记。

```
■         if (!player.inTunnel) {
■             switchMovement(MOVESTOP, player);
■             /** uncommented in iteration #6
■             setCurrentRegion(player);
```



```
■          /*** end uncommented in iteration #6
■          //once player gets to center, check the enemy chase AI
■          }
```

## 给敌坦克修改 update 和 render 函数

敌坦克的更新和玩家的相同。它们可以像玩家一样移动 maze，甚至是利用游戏场景外的上，下，侧面的“时空隧道”。要渲染敌坦克，我们首先要更新 nextX 和 nextY 值。然后，如果坦克在一个隧道里，我们要检测它是否在隧道里。最后，我们要更新 currRow 和 currCol 属性，并调用敌坦克的 updateTile 函数。我们通过循环 enemyTanks 数组来给每一个敌坦克做这些内容。

这并没有完全重写 update 函数。你需要给已放在场景中的玩家渲染在当前的 update 函数下面添加这些代码。这里是将会控制敌坦克的新添加的 update 函数：

```
■          /*** added iteration #6
■          var enemyLength:int = enemyList.length-1;
■          for (var ctr:int = enemyLength; ctr >= 0; ctr--) {
■              tempEnemy = enemyList[ctr];
■
■              tempEnemy.nextX = tempEnemy.x +
tempEnemy.dx*tempEnemy.moveSpeed;
■              tempEnemy.nextY = tempEnemy.y + tempEnemy.dy *
tempEnemy.moveSpeed;
■
■              //check to see is enemy off side of screen then set in tunnel
■              if (tempEnemy.y <yMin || tempEnemy.y>yMax || #
■                  tempEnemy.x<xMin || tempEnemy.x >xMax){
■                  tempEnemy.inTunnel = true;
■              }else {
■                  tempEnemy.inTunnel = false;
■              }
■
■              if (tempEnemy.inTunnel) {
■                  switch(tempEnemy.currentDirection) {
■
■                      case MOVE_UP:
```



```

    if (tempEnemy.nextY == yTunnelMin){
        tempEnemy.nextY=yTunnelMax
    }else if (tempEnemy.nextY == yMax) {
        tempEnemy.inTunnel = false;
    }
    break;

case MOVE_DOWN:
    if (tempEnemy.nextY == yTunnelMax){
        tempEnemy.nextY = yTunnelMin;
    }else if (tempEnemy.nextY == yMin) {
        tempEnemy.inTunnel = false;
    }
    break;

case MOVE_LEFT:
    if (tempEnemy.nextX == xTunnelMin){
        tempEnemy.nextX = xTunnelMax;
    }else if (player.nextX == xMax) {
        tempEnemy.inTunnel = false;
    }
    break;

case MOVE_RIGHT:
    if (tempEnemy.nextX == xTunnelMax) {
        tempEnemy.nextX = xTunnelMin;
    }else if (player.nextX == xMin) {
        tempEnemy.inTunnel = false;
    }
    break;

case MOVE_STOP:
    //trace("stopped");
    break;
}

tempEnemy.currRow = tempEnemy.nextY / tileWidth;
tempEnemy.currCol = tempEnemy.nextX / tileHeight;
```



```
■         tempEnemy.updateCurrentTile();  
■     }  
■     /** end added iteration #6  
■
```

记住这些代码需要添加到现存的 update 函数中。我们没有显示闭括号 ( ) 来结束函数是因为它已经存在于现在的代码中了。

如你所见，要更新敌人和移动他们的代码和玩家的代码相同。

下面是要添加到 render 函数中的代码，用来控制敌坦克。敌坦克的 render 函数代码和玩家的非常相似。最大的不同是包含了两个调用，就是用来为敌坦克执行 AI 的。这两个调用是 chaseObject 和 checkLineOfSight 函数。添加下面的代码到已存在的 render 函数下面：

```
■         /** added in iteration #6  
■         var enemyLength:int = enemyList.length-1;  
■         for (var ctr:int = enemyLength; ctr >= 0; ctr--) {  
■  
■             tempEnemy = enemyList[ctr];  
■             tempEnemy.x = tempEnemy.nextX;  
■             tempEnemy.y = tempEnemy.nextY;  
■             setCurrentRegion(tempEnemy);  
■  
■             if (checkCenterTile(tempEnemy)) {  
■  
■                 //trace("enemy @ center tile");  
■                 if (!tempEnemy.inTunnel) {  
■                     switchMovement(MOVE_STOP, tempEnemy);  
■                     chaseObject(player, tempEnemy);  
■                 }  
■             }  
■  
■  
■         //should enemy fire  
■         checkLineOfSight(player, tempEnemy);  
■
```



```
■ tempEnemy.renderCurrentTile();  
■ }  
■ /** end added in iteration #6  
■ Again, we have not shown the final } for the function, because it should already exist  
in the code.  
■
```

这里还是没有闭括号}, 原因同上。

## 给敌坦克 AI 添加新的方法

接下来要给它添加两个方法, 来让敌坦克更聪明一点。我们还没有给敌坦克添加一个比较强的 AI, 现在的 AI 仅仅可以小小的挑战一下你。我们称这个为“正合适的 AI”。一旦你看到了它是如何工作的, 就可以把它修改和扩展的更聪明了。

### 追上你!

如果你在敌坦克的同一个区域, 我们希望敌坦克死命的追着你。我们要单独的实现每一个敌坦克的此目标, 通过计算你到敌坦克之间的块的数量。左右的不同块会调用 horizontalDiff, 上下的不同块调用 verticalDiff。见图 7-6。

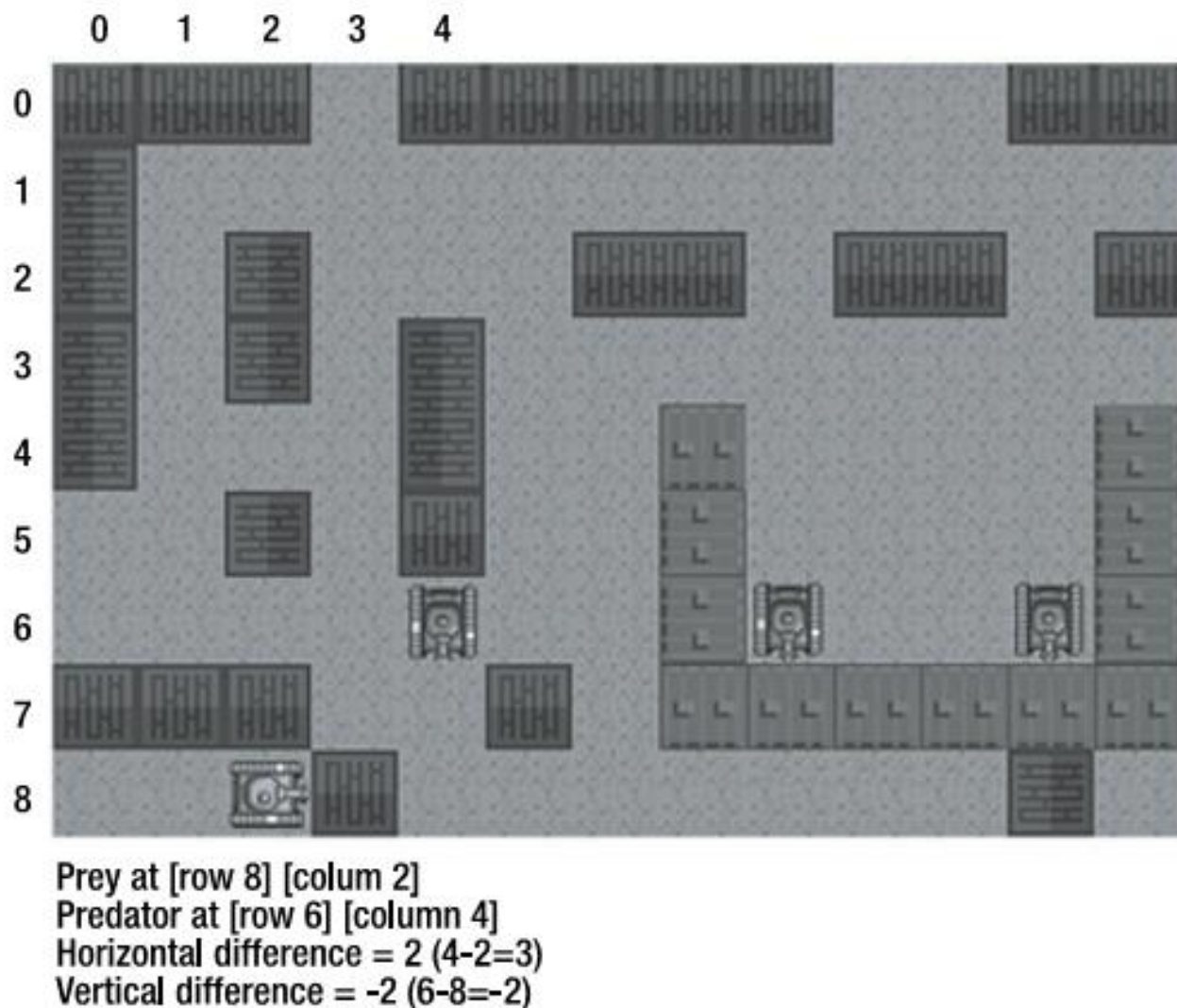


图 7-6 追逐示例

我们会创建一个叫做 `chaseObject` 的方法，它可以被其他基于块结构的游戏重用。这个方法有两个参数（都是 `TileByTileBlitSprite` 实例）。第一个对象是猎物，第二个是掠食者。如果他们在同一块中，掠食者就会追逐他个猎物。如果他们在同一区域，

我们尝试着给掠食者创建一个移动可能性的表单。这个表单里的可能的运动方向会被优先选用，掠食者就必须移动去寻找他的猎物。我们在块位置检测水平位置上的异样并把它存放在 `horizontalDiff` 变量里，把垂直方向上的不同存放到 `verticalDiff` 变量里。

我们会用 `Math` 方法的绝对值来比较 `horizontalDiff` 和 `verticalDiff`。我们希望这个追赶者（掠食者）



向比较得到的那个块数最少的方向移动。当我们试图比较方向时有一个问题，因为我们不能恰当的比较负值，这个结果是用简单的 `math` 得来的正值（绝对值）。例如，这里是一个类似于图 7-6 的例子的代码所需要的步骤：

- 猎物在列 2 行 8。
- 捕食者在列 4 行 6。
- `predator.currCol` 减去 `prey.currCol` 等于 2。
- 如果猎物在行 8，捕食者在行 6，那么 `pred.currRow` 减去 `prey.currRow` 等于 -2。
- 所有方向都是 2，负数结果来自于那个相减的命令。

当我们比较这些来观察哪一个更小（就是哪个方向上猎物更接近捕食者），我们必须用 `Math.abs`（绝对值）方法。这么做是因为 -2 永远小于 2 除非我们比较绝对值。要想比较两者之间块距离远近，-2 和 2 的距离是相同的。再次强调，我们只在意精灵的坐标位置之间的不同块的数量；不管计算结果是正或负。

当我们已经选择了垂直或水平移动，然后就要选择敌坦克要往哪个具体方向移动（水平的左或右还是垂直的上或下）。

首先，我们要看看代码，然后讨论这个函数的要点。

## chaseObject 方法

下面是 `chaseObject` 方法的代码：

```
■ private function chaseObject(pre:TileByTileBlitSprite,#
■ predator:TileByTileBlitSprite):void {
■
■ //trace("chase");
■
■ if (pre.currentRegion == predator.currentRegion) {
■     moveDirectionsToTest = [];
```



```
var horizontalDiff:int = predator.currCol - prey.currCol;
var verticalDiff:int = predator.currRow - prey.currRow;

if (Math.abs(verticalDiff) < Math.abs(horizontalDiff)) {
    if (verticalDiff > 0) {
        //trace("AI UP");
        moveDirectionsToTest.push(MOVE_UP);
        moveDirectionsToTest.push(MOVE_DOWN);
    }
    else if (verticalDiff < 0) {
        //trace("AI DOWN");
        moveDirectionsToTest.push(MOVE_DOWN);
        moveDirectionsToTest.push(MOVE_UP);
    }
}

if (horizontalDiff > 0) {
    //trace("AI LEFT");
    moveDirectionsToTest.push(MOVE_LEFT);
    moveDirectionsToTest.push(MOVE_RIGHT);
}
else if (horizontalDiff < 0) {
    //trace("AI RIGHT");
    moveDirectionsToTest.push(MOVE_RIGHT);
    moveDirectionsToTest.push(MOVE_LEFT);
}
}

if (Math.abs(horizontalDiff) < Math.abs(verticalDiff)) {
    if (horizontalDiff > 0) {
        //trace("AI LEFT");
        moveDirectionsToTest.push(MOVE_LEFT);
        moveDirectionsToTest.push(MOVE_RIGHT);
    }
    else if (horizontalDiff < 0) {
        //trace("AI DOWN");
        moveDirectionsToTest.push(MOVE_RIGHT);
        moveDirectionsToTest.push(MOVE_LEFT);
    }
}
if (verticalDiff > 0) {
```



```

    //trace("AI UP");
    moveDirectionsToTest.push(MOVE_UP);
    moveDirectionsToTest.push(MOVE_DOWN);

}
else if (verticalDiff < 0) {
    //trace("AI DOWN");
    moveDirectionsToTest.push(MOVE_DOWN);
    moveDirectionsToTest.push(MOVE_UP);
}
}

//make an educated guess
if (Math.abs(horizontalDiff) == Math.abs(verticalDiff)) {
    trace("AI Random");
    if (int(Math.random() * 2) == 0) {
        //vertical
        if (verticalDiff > 0) {
            //trace("AI UP");

            moveDirectionsToTest.push(MOVE_UP);
            moveDirectionsToTest.push(MOVE_DOWN);
        }
        else if (verticalDiff < 0) {
            //trace("AI DOWN");

            moveDirectionsToTest.push(MOVE_DOWN);
            moveDirectionsToTest.push(MOVE_UP);
        }
    }
    else {
        //horizontal
        if (horizontalDiff > 0) {
            //trace("AI LEFT");

            moveDirectionsToTest.push(MOVE_LEFT);
            moveDirectionsToTest.push(MOVE_RIGHT);
        }
        else if (horizontalDiff < 0) {
            //trace("AI DOWN");

            moveDirectionsToTest.push(MOVE_RIGHT);
        }
    }
}
```



```
■          moveDirectionsToTest.push(MOVE_LEFT);
■      }
■  }
■  }
■
■      if (horizontalDiff == 0 && verticalDiff == 0) {
■          //trace("AI STOP");
■          moveDirectionsToTest = [MOVE_STOP];
■      }
■
■      //as a final move for all, push in a MOVE_STOP
■      moveDirectionsToTest.push(MOVE_STOP);
■
■      //moveDirectionsToTest should now have a list of moves in it.
■      //loop though those, check them and set the dx and dy of the enemy.
■
■      var moveFound:Boolean = false;
■      var movePtr:int = 0;
■      var move:int;
■
■      while (!moveFound) {
■          move = moveDirectionsToTest[movePtr];
■          if (move==MOVE_UP && predator.inTunnel==false) {
■
■              if (checkTile(MOVE_UP, predator)) {
■
■                  switchMovement(MOVE_UP, predator );
■                  moveFound = true;
■              }
■          }
■
■          if (move==MOVE_DOWN && predator.inTunnel==false) {
■
■              //trace("player.currentDirection=" + player.currentDirection);
■              if (checkTile(MOVE_DOWN, predator)) {
■
■                  switchMovement(MOVE_DOWN, predator );
■                  moveFound = true;
■              }
■          }
■      }
```



```

    if (move==MOVE_RIGHT && predator.inTunnel==false) {
        if (checkTile(MOVE_RIGHT, predator)) {

            switchMovement(MOVE_RIGHT, predator );
            moveFound = true;
        }
    }

    if (move==MOVE_LEFT && predator.inTunnel==false) {
        if (checkTile(MOVE_LEFT,predator)) {

            switchMovement(MOVE_LEFT, predator );
            moveFound = true;
        }
    }

    if (move==MOVE_STOP && predator.inTunnel==false) {
        switchMovement(MOVE_STOP, predator );
        moveFound = true;
    }

    movePtr++;
    if (movePtr == moveDirectionsToTest.length) {
        switchMovement(MOVE_STOP, predator );
        moveFound = true;
    }
}
}
```

## 这就是 chaseObject 方法的要点：

- 首先，这个方法有两个参数，都是 TileByTileBlitSprites。第一个是猎物，第二个是追逐者。如果他们处在同一个区域，追逐者就会追赶猎物。
- 方法的第一部分搜索方向来填充 moveDirectionsToTest 数组。这个移动的数组会用来在方法的



第二部分搜寻一个要走的方向，当然这个方向不能是墙了。

- horizontalDiff 和 verticalDiff 的本地变量计算着捕食者和猎物 水平的和垂直的（块的）不同。

- 两组

状态来比较 verticalDiff 和 horizontalDiff 变量的绝对值，所以我们可以评估出两者在哪

个方向上。更接近。我们只关心哪个数目（verticalDiff 和 horizontalDiff）更小。

- if 的第一组状态是垂直方向上的（来决定 verticalDiff 是否比 horizontalDiff 更小）。如果 verticalDiff 小于 0，猎物就在捕食者的上方。如果 verticalDiff 大于 0，就在下方。如果上方时最近的方向，我们按下 MOVE\_UP 然后 MOVE\_DOWN 到上方。如果下方是最近的方向，按下 MOVE\_DOWN

然后 MOVE\_UP 到下方。我们为什么要这么做？这就是“最合适的 AI”所体现的。我们想让捕食者

尝试一连串的移动。目的就是让敌坦克在他们在块中心的每一帧发现最少一个方向去运动（或停止）。很明显，即使捕食者非常想去上方，但是上方却是一堵墙，就只能让他下拐了。可见，如果 verticalDiff 比 horizontalDiff 小，我们就会在垂直方向移动，如果 verticalDiff 大于 0，就让捕食者向上运动。如果他不能向上运动，就让他试着向下移动最为第二佳选择。

- 代码要做的下一个任务就是去看看水平方向运动的哪一个（方向）被添加到了 moveDirectionsToTest。数组里。第一组的两个方向已经是 MOVE\_UP 和 MOVE\_DOWN（在这个命令里 不需要），所以就去填写 moveDirectionsToTest 数组，再去看看 horizontalDiff 是否大于 0。MOVE\_LEFT 和 MOVE\_RIGHT 被基础的添加在这个测试里。

- if 的第二组状态要测试 horizontalDiff 是否小于 verticalDiff。这种情况会在捕食者在水平方向上更接近猎物时出现。它的工作方式跟 verticalDiff 更小时很相似，但是优先处理水平的运动。



- 如果 horizontalDiff 和 verticalDiff 都是 0，即追赶着在猎物的正上方，那么捕猎者就不动。
- 如果 horizontalDiff 和 verticalDiff 的绝对值相等，就选择一个基于随机值的方向，并做一个有把握的猜测。随机值在水平和垂直上选择。
- MOVE\_STOP 总是添加到 moveDirectionsToTest 数组的最后。
- 代码的最后部分用一个循环来为追赶者（敌坦克）选择一个合适的方向。如果找不到要运动的方向，敌坦克就会保持停止。

这个 AI 有诸多限制，但是对于给玩家一个小小的挑战已经足够了。也可以扩展它为不允许敌坦克向来时的方向运动（除非没有其他的路可走了）。这可以预防一些坦克的后转。我们不会在此章中涉及到它，但如果你现在直到运动表单时如何工作的，你可以根据你的需要加入一些逻辑了。有一些非常好的路径寻找算法，像 A\*，也可以在两者之间找到最短的距离。

## checkLineOfSight 方法

checkLineOfSight 方法用来准予敌人转向和向玩家开火。什么是瞄准线(line of sight)？只有当视线没有被挡住的时候敌坦克才会转向并向玩家开火。这是通过寻找玩家与敌人的关系计算得来的，然后循环他们笔直之间的每一个块（只有水平或垂直时）。在这个循环中，如果我们攻击墙或离开场景，敌坦克就不会向玩家开火。如果我们和玩家到了同一个块，敌坦克就会向玩家开火。

一个 Level 的实例属性叫做 enemyIntelligence 用来决定敌人的视线内但和玩家不在同一区域时的可能性。这个值在 0-100 之间。数值越大，敌人转向玩家开火的几率就越大，如果在敌人视线内的话。我们还没有给敌人实现 level-specific 变量的任何一个（比如 enemyIntelligence），但是等一会在本章的后面当我们清理最后的游戏时，会添加这些。现在，我们给 enemyIntelligence 用一个值为 50 的占位符，这



个变量的名字就在下面得代码中：

```
■ private function checkLineOfSight(pre:TileByTileBlitSprite, #
■ predator:TileByTileBlitSprite):void {
■
■ //rotation reference
■ //up=0;
■ //right=90;
■ //down=180;
■ //left=-90;
■
■ var testDX:int;
■ var testDY:int;
■ var testRotation:int;
■ var checkCol:int;
■ var checkRow:int;
■ var difference:int;
■ var differenceCtr:int;
■ var act:Boolean = false;
■
■ //1. test if they are in the same col or row
■ //2. if in same col (horizontalDiff), check if verticalDiff is pos or neg
■ //3. if in same row (verticalDiff), check if horizontalDiff is pos or neg
■
■ /**placeholder
■ var enemyIntelligence:int = 50;
■
■
■ if (pre.currentRegion == predator.currentRegion) {
■ act = true;
■ }else if (int(Math.random() * 100) < enemyIntelligence) {
■ //trace("act based on intel");
■ //if not in same region, then turn toward player
■ //first be sure to turn toward the player
■
■ var horizontalDiff:int = predator.currCol - pre.currCol;
■ var verticalDiff:int = predator.currRow - pre.currRow;
■ if (verticalDiff >0) {
```



```

    //trace("AI UP");
    predator.rotation = 0;

}

}else if (verticalDiff < 0) {
    //trace("AI DOWN");
    predator.rotation = 180;

}

}else if (horizontalDiff > 0) {
    //trace("AI LEFT");
    predator.rotation = -90;

}

}else if (horizontalDiff < 0) {
    //trace("AI RIGHT");
    predator.rotation = 90;
}
act = true;
}

if (act) {

    if (predator.currCol == prey.currCol) {
        //trace ("same col");
        difference = Math.abs(predator.currRow - prey.currRow);

        if (predator.currRow < prey.currRow) {
            testDX = 0;
            testDY = 1;
            testRotation = 180;
        }else if (predator.currRow > prey.currRow) {
            testDX = 0;
            testDY = -1;
            testRotation = 0;
        }else {
            testDX = 0;
            testDY = 0;
            testRotation = 99;
        }
    }else if (predator.currRow == prey.currRow) {
        difference = Math.abs(predator.currCol- prey.currCol);
    }
}
```



```
// trace ("same row");

if (predator.currCol < prey.currCol) {
    testDX = 1;
    testDY = 0;
    testRotation = 90;
}else if (predator.currCol > prey.currCol) {
    testDX = -1;
    testDY = 0;
    testRotation = -90;
}else {
    testDX = 0;
    testDY = 0;
    testRotation = 99;
}
}else {
    difference = 0;
}

checkCol = predator.currCol;
checkRow = predator.currRow;

for (differenceCtr = 0; differenceCtr <= difference; differenceCtr++) {
    checkCol += testDX;
    checkRow += testDY;

    if (checkCol < 0 || checkCol == mapColumnCount) {
        break;
        //trace("col hit border");
    }

    }else if (checkRow < 0 || checkRow == mapRowCount) {
        break;
        //trace("row hit border");
    }

    }else if (tileSheetData[levelTileMap[checkRow][checkCol]] == TILE_WALL)
    {
        //trace("hit wall");
        break;
    }

    }else if (checkCol == player.currCol && checkRow == player.currRow) {
```



```

■          //if predator is facing the player then fire
■          if (predator.rotation == testRotation) {
■              fireMissileAtPlayer(predator);
■          }
■          break;
■
■          }else {
■              //trace("hit nothing");
■          }
■      }
■  }
■
■  }
■
■  /** end added in iteration #6

```

## CheckLineOfSight 方法里的要点：

- 捕猎者和猎物被通过就像他们在 chaseObject 方法中。
- 如果他俩已经在同一个区域了，本地的 act 变量被设置为 true 来迫使等会的代码测试视线。
- 如果他俩不再同一区域， enemyIntel 就被用来决定捕猎者对象是否要对猎物进行视线测试。

如果他们在同一区域， act 变量就会为 true，或者随机值小于 enemyIntel。同样，捕猎者被转向到面向猎物，所以他看起来聪明。

- 如果 act 是 true，我们首先就要检测捕猎者和猎物在同一纵列。如果是，我们就设置本地 difference 变量为两者块距离的绝对值。然后用嵌套的 if : else 结构来寻找我们需要测试的是哪个方向。本地的 testDX 和 testDY 变量标明每个方向的块的数量。 TestRotation 用来旋转捕食者到假设的玩家所在方向开火，如果测试为真的话。
- 如果他俩的当前列不同，就要像第三步一样检测行。
- 最后，如果他们不再同一行或列，我们设置 difference 为 0。这将预防下一部分的循环并结束捕食者的视线测试。
- 循环从 0 到 difference（两者之间的块位置的不同在第 4 步算出）。在每一个循环迭代器中，我



们

都添加 testDX 值到 checkCol 和 testDY 到 checkRow。如果敌人攻击玩家的当前块—没有被墙挡

住

也没有离开场景，他就会像玩家开火（当然也会向玩家的方向转向）。

## fireMissileAtPlayer 方法

我们现在添加 fireMissileAtPlayer 作为占位符，直到我们在下一部分中添加射弹。它会包含一个 trace 语句，这样我们就可以看到敌坦克 AI 的逻辑像玩家开火。

```
■      private function fireMissileAtPlayer(enemySprite:TileByTileBlitSprite):void {  
■          trace("fire at player");  
■      }
```

## 添加到变量定义

我们仍然还没有添加 moveDirections 数组，那个给这个迭代器用在 chaseObject 函数到变量定义区的数组。添加下面一行到变量定义区：

```
■      private var moveDirectionsToTest:Array;
```

## 测试迭代器 6

当你测试这个迭代器，你应该会看到所有的敌坦克都被放到场景中了（见图 7-7）。他们应该会朝向玩家移动如果他们在同一区域的话，同样也会转向和开枪。你会注意到这会有很多跟踪状态。如果这汤尼的 IDE 活 Flash Develop 变慢的话，就去掉一个再试一次。

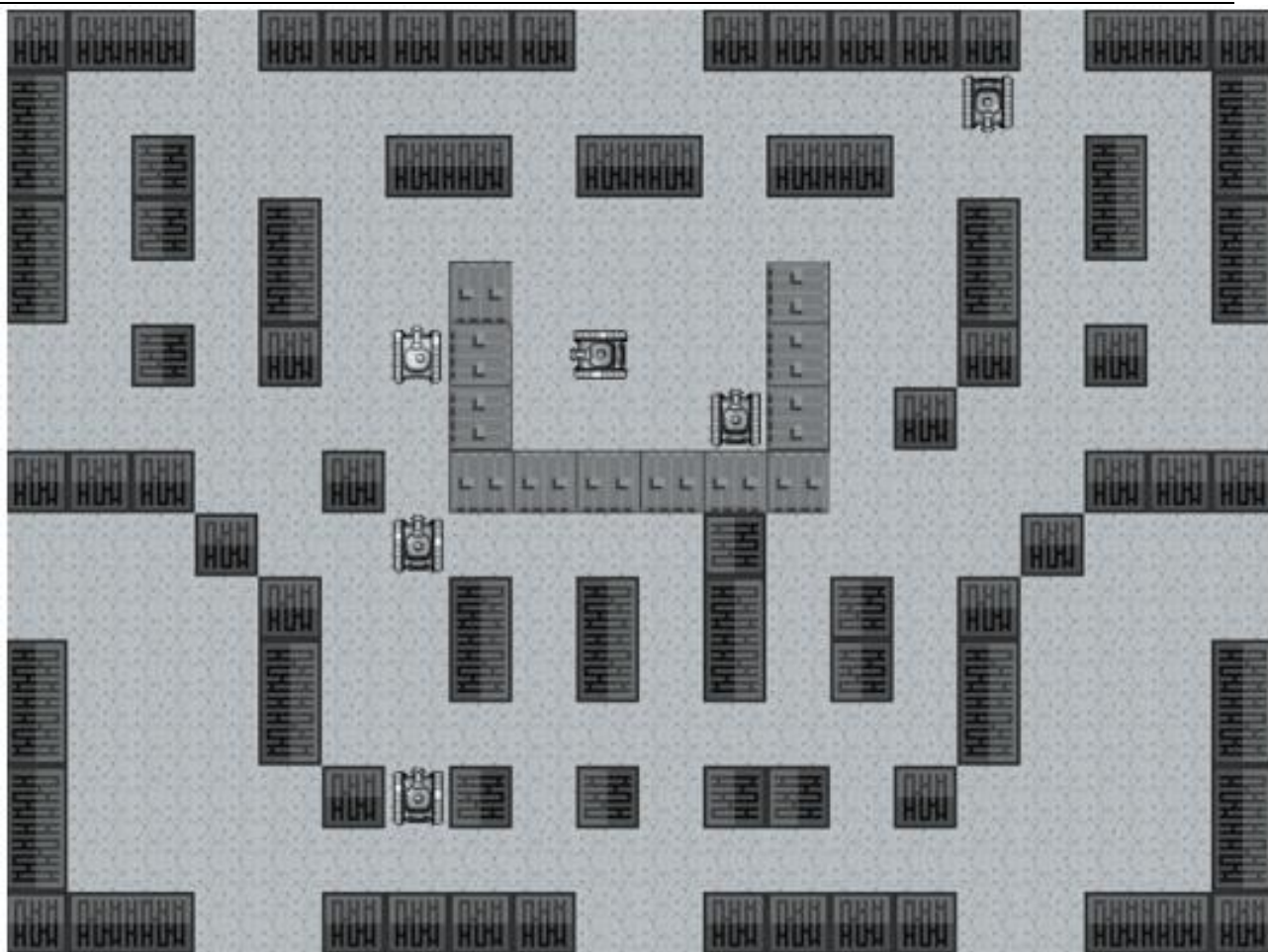


图 7-7 迭代器 6 的游戏场景，敌坦克正在向玩家移动

## 融入到框架中

在我们将玩家和敌军坦克发射的子弹,爆炸,关卡(levels),弹药和其他可拾取物体添加到游戏之前,是时候将游戏添加到框架中了.因此我们将在游戏包结构中的某个位置创建一个 Main.as 文件.我们还将修改在第六章中创建的 Library 类和 TileSheetDataXML 类.

## 继承到框架类 Main.as 中

和往常一样,我们会在开始部分告诉你这个新类在 FLash IDE 中的路径:



/source/projects/notanks/flashIDE/com/efg/games/notanks]Main.as

和 Flex SDK (使用 Flash Develop):

/source/projects/notanks/flexSDK/src/com/efg/games/notanks/Main.as

我们不会讲述这个类的每个细节,因为上一章中我们已经讲述过了.我们已经添加了一些新的,我们会高亮显示出例如那些新的,独特的用来处理屏幕上声音播放的代码.

```
■ package com.efg.games.notanks
■ {
■
■     import flash.text.TextFormat;
■     import flash.text.TextFormatAlign;
■     import flash.geom.Point;
■
■     import com.efg.framework.FrameWorkStates;
■     import com.efg.framework.GameFrameWork;
■     import com.efg.framework.BasicScreen;
■     import com.efg.framework.ScoreBoard;
■     import com.efg.framework.SideBySideScoreElement;
■     import com.efg.framework.SoundManager;
■
■
■     public class Main extends GameFrameWork {
■
■
■         //custom score board elements
■         public static const SCORE_BOARD_SCORE:String = "score";
■         public static const SCORE_BOARD_AMMO:String = "ammo";
■         public static const SCORE_BOARD_TANKS:String = "tanks";
■         public static const SCORE_BOARD_HEALTH:String = "health";
■
■         //custom sounds
■         public static const SOUND_ENEMY_FIRE:String = "enemyfire";
■         public static const SOUND_EXPLODE:String = "explode"
■         public static const SOUND_PLAYER_EXPLODE:String = "playerexplode";
■         public static const SOUND_PLAYER_FIRE:String = "playerfire";
```



```
■ public static const SOUND_PLAYER_MOVE:String = "playermove";
■ public static const SOUND_PICK_UP:String = "pickup";
■ public static const SOUND_GOAL:String = "goal";
■ public static const SOUND_HIT:String = "hit";
■ public static const SOUND_MUSIC:String = "music";
■ public static const SOUND_HIT_WALL:String = "hitwall";
■ public static const SOUND_LIFE:String = "soundlife";
■
■
■ public function Main() {
■     init();
■ }
■
■ override public function init():void {
■     game = new NoTanks();
■     game.y = 20;
■     setApplicationBackGround(640, 500, false, 0x000000);
■
■
■ //add score board to the screen as the seconf layer
■     scoreBoard = new ScoreBoard();
■     addChild(scoreBoard);
■
■     scoreBoardTextFormat= new TextFormat("_sans", "11", "0xffffffff", "true");
■
■     scoreBoard.createTextElement(SCORE_BOARD_SCORE, new
■         SideBySideScoreElement(80, 5, 20, "Score", scoreBoardTextFormat, 25,
"0",
■         scoreBoardTextFormat));
■
■     scoreBoard.createTextElement(SCORE_BOARD_AMMO, new
■         SideBySideScoreElement(180, 5, 20, "Ammo", scoreBoardTextFormat, 25,
"0/0", scoreBoardTextFormat));
■
■     scoreBoard.createTextElement(SCORE_BOARD_TANKS, new
■         SideBySideScoreElement(280,5, 20, "Tanks", scoreBoardTextFormat, 25,
"0%", scoreBoardTextFormat));
■
■     scoreBoard.createTextElement(SCORE_BOARD_HEALTH, new
■         SideBySideScoreElement(380, 5, 20, "Health", scoreBoardTextFormat, 25,
```



```
        "0%", scoreBoardTextFormat));  
  
    //screen text initializations  
    screenTextFormat = new TextFormat("_sans", "16", "0xffffffff", "false");  
    screenTextFormat.align = flash.text.TextFormatAlign.CENTER;  
    screenButtonFormat = new TextFormat("_sans", "12", "0x000000", "false");  
  
    titleScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE,640,500,  
        false,0x0000dd );  
  
    titleScreen.createOkButton("Play", new Point(250, 250), 100, 20,  
        screenButtonFormat, 0x000000, 0xff0000,2);  
  
    titleScreen.createDisplayText("坦克大战", 120,new Point(245,150),  
        screenTextFormat);  
  
    instructionsScreen = new BasicScreen(FrameWorkStates.  
        STATE_SYSTEM_INSTRUCTIONS,640,500,false,0x0000dd);  
  
    instructionsScreen.createOkButton("Start", new Point(250, 250), 100, 20,  
        screenButtonFormat, 0x000000, 0xff0000,2);  
  
    instructionsScreen.createDisplayText("Find the treasure.\nDestroy the  
tanks!\nArrows and Space",250,new Point(180,150),screenTextFormat);  
  
    gameOverScreen = new  
BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER,  
        640,500,false,0x0000dd);  
  
    gameOverScreen.createOkButton("Restart", new Point(250, 250), 100,  
        20,screenButtonFormat, 0x000000, 0xff0000,2);  
  
    gameOverScreen.createDisplayText("Game Over",100,new Point(250,150),  
        screenTextFormat);  
  
    levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN,  
        640, 500, true, 0xaa0000);  
        levelInText = "Level ";  
    levelInScreen.createDisplayText(levelInText,100,new Point(250,150),  
        screenTextFormat); //set initial game state
```



```
■ switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
■
■ //wait
■ waitTime = 30;
■
■ //sounds
■ /*** flex SDK
■ soundManager.addSound(SOUND_ENEMY_FIRE,new Library.SoundEnemyFire);
■ soundManager.addSound(SOUND_EXPLODE, new Library.SoundExplode);
■ soundManager.addSound(SOUND_PLAYER_EXPLODE,new
Library.SoundPlayerExplode);
■ soundManager.addSound(SOUND_PLAYER_FIRE,new Library.SoundPlayerFire);
■ soundManager.addSound(SOUND_PLAYER_MOVE,new
Library.SoundPlayerMove);
■ soundManager.addSound(SOUND_PICK_UP,new Library.SoundPickUp);
■ soundManager.addSound(SOUND_GOAL,new Library.SoundGoal);
■ soundManager.addSound(SOUND_HIT,new Library.SoundHit);
■ soundManager.addSound(SOUND_MUSIC,new Library.SoundMusic);
■ soundManager.addSound(SOUND_HIT_WALL,new Library.SoundHitWall);
■ soundManager.addSound(SOUND_LIFE, new Library.SoundLife);
■ /*** end flex sdk
■
■ /***Flash IDE
■ */
■ soundManager.addSound(SOUND_ENEMY_FIRE,new SoundEnemyFire);
■ soundManager.addSound(SOUND_EXPLODE, new SoundExplode);
■ soundManager.addSound(SOUND_PLAYER_EXPLODE,new
SoundPlayerExplode);
■ soundManager.addSound(SOUND_PLAYER_FIRE,new SoundPlayerFire);
■ soundManager.addSound(SOUND_PLAYER_MOVE,new SoundPlayerMove);
■ soundManager.addSound(SOUND_PICK_UP,new SoundPickUp);
■ soundManager.addSound(SOUND_GOAL,new SoundGoal);
■ soundManager.addSound(SOUND_HIT,new SoundHit);
■ soundManager.addSound(SOUND_MUSIC,new SoundMusic);
■ soundManager.addSound(SOUND_HIT_WALL,new SoundHitWall);
■ soundManager.addSound(SOUND_LIFE, new SoundLife);
■ */
■
■
■
```



```
■ //create timer and run it one time
■ frameRate = 30;
■ startTimer();
■ }
■
■ override public function systemTitle():void {
■     soundManager.playSound(SOUND_MUSIC, false,999, 0, 1);
■     super.systemTitle();
■ }
■
■ override public function systemNewGame():void {
■     soundManager.stopSound(SOUND_MUSIC);
■     super.systemNewGame();
■ }
■ }
■ }
```

该应用的背景和游戏中 Sprite 的位置

现在我们将要为该应用设置背景颜色并设置游戏中 Sprite 在屏幕上的位置.

1.在变量定义的段落,我们将该应用的大小设置为 640x500 pixels.我们已经在顶端为得分板留出了 20 像素.

```
setApplicationBackGround(640, 500, false, 0x000000);
```

2.我们已经将游戏实例向下移动,即将其 y 属性设置为 20,为的是让屏幕上端可以容纳下得分板.

```
game.y = 20;
```

## 得分板

我们分别将我们想让用户看到的 SCORE, AMMO, TANKS 和 HEALTH 四个元素添加到得分板上.这些常量 将会被添加到变量定义的部分并且 createTextElement 将会在 init 函数中被调用。

下面就是我们将要添加的 4 个新元素:



Score:玩家得分

Ammo: 玩家弹药剩余量(每个坦克的初始量是 50)

Health:玩家坦克生命点数剩余量(初始值是 5)

Tanks: 玩家坦克(命)剩余量

## 屏幕

我们为 4 个屏幕 (titleScreen, instructionsScreen, gameOverScreen, 和 levelInScreen)添加了文字,位置和按钮(如果可能)。

## 声音

我们在变量定义的部分添加了所有的声音(名)的常量并且 addSound 函数会调用 init 函数.注意 在 Flex SDK 的代码中,我们已经将 Library 类名添加到所有声音代码的前面.这是 Flash IDE 和 Flex SDK 代码的一个很重要的不同之处,我们会在下一节融合声音时具体的解释这一点。

## 为 Sounds 而重写(Overrides)的新函数

在 GameFrameWork.as 文件中我们有 2 个最新重写(Overrides)的函数.systemTitle 和 systemNewGame 函数 将被 建立 并用于 处理 声音 SOUND\_MUSIC 的 播放 和 停止 . 在 GameFrameWork.as 中,这一过程会在第一次调用SoundManager类的playSound 或者 stopSound以及他们在父类中版本的函数时完成。

## 完成 Library 类

声音是一个比较特殊例子,因为他们需要有实体的声音,所以你必须提前创建好。在第 10 和 11 章中,我们将会 在 Flex SDK AS3 的 application 中 嵌入(使用 Flex 的 embed) MP3 格式的声音。现在,我们使用



和第 4,5 章中同样的方法。你需要将所有的声音嵌入到一个 fla 中,并将其导出名设置成和 Library.as 文件中的名字一致(下面就会讲到),然后将其导出为 noTanks\_assets.swf。你需要将这个文件放到项目的 assets 文件夹中。如果你是使用 Flash IDE 则不需要做这这些。在这种情况下,我们只需要简单的将其导入并将其连接标识名(linkage identifiers)设置为类中的名字即可。

下面的是这个文件在框架中的位置:

```
/source/projects/notanks/flexSDK/assets/noTaks_assets.swf
```

为了可以处理每一个我们需要的声音,我们将新素材更新到 Library.as 中.这个更新文件的位置将在:

```
/source/projects/notanks/flexSDK/src/com/efg/games/notanks/Library.as
```

下面的就是新的 Library.as 文件把 noTanks\_assets.swf 当做声音源来嵌入声音的代码:

```
■ package com.efg.games.notanks
■ {
■     import flash.display.Bitmap;
■     import flash.display.BitmapData;
■
■     public class Library {
■
■         [Embed(source='.././../assets/tanks_sheet.png')]
■         public static const TankSheetPng:Class;
■
■         [Embed(source = ".././../assets/noTanks_assets.swf",
■             symbol="soundEnemyFire")]
■         public static const SoundEnemyFire:Class;
■
■         [Embed(source = ".././../assets/noTanks_assets.swf",
■             symbol="soundExplode")]
■         public static const SoundExplode:Class;
```



```
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf",  
■     symbol="soundPlayerExplode")]  
■ public static const SoundPlayerExplode:Class;  
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf",  
■     symbol="soundPlayerFire")]  
■ public static const SoundPlayerFire:Class;  
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf",  
■     symbol="soundPlayerMove")]  
■ public static const SoundPlayerMove:Class;  
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf",  
■     symbol="soundPickUp")]  
■ public static const SoundPickUp:Class;  
■ [Embed(source = "../..../assets/noTanks_assets.swf", symbol="soundGoal")]  
■ public static const SoundGoal:Class;  
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf", symbol="soundHit")]  
■ public static const SoundHit:Class;  
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf",  
■     symbol="soundMusic")]  
■ public static const SoundMusic:Class;  
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf",  
■     symbol="soundHitWall")]  
■ public static const SoundHitWall:Class;  
■  
■ [Embed(source = "../..../assets/noTanks_assets.swf", symbol="soundLife")]  
■ public static const SoundLife:Class;  
■ }  
■ }
```

我们使用 SFXR 工具来创建所有的声音.在第 4,5 章已经有此工具的详细介绍。通过使用 Sony Acid 的 Magixloop library 来创建音乐。我们将音乐保存为 wav 格式的文件,所以当被导出为游戏的 SWF 时它就有了循环播放的属性。



## 搞定 level.as 和 Level1.as 文件

我们在上一章中创建了 Level.as 文件.我们需要给关卡识别数据添加 2 个新属性.

第一个新属性叫做 enemyShotDelay.它代表了敌人多次射击玩家时的射击时间间隔.间隔越短游戏关卡的难度将越高.

第二个新属性叫做 enemyHealthPoints.它代表了在敌军被玩家摧毁前可以承受多少次弹药的打击.

这个文件应该已经存在于:

```
/source/projects/notanks/flexSDK/src/com/efg/games/notanks/Level.as
```

在该文件中为该变量添加一行新代码.下面是 Level.as 文件更新后的全部代码:

```
■ package com.efg.games.notanks
■ {
■ /**
■  * ...
■  * @author Jeff Fulton
■  */
■     public class Level{
■
■         public var backGroundMap:Array;
■         public var spriteMap:Array;
■         public var backGroundTile:int;
■         public var enemyIntelligence:int;
■         public var enemyShotSpeed:Number;
■         public var ammoPickUp:int;
■
■         /**added in chapter 7
■     public var enemyHealthPoints:Number;
■         public var enemyShotDelay:int;
■
■         public function Level() {
■         }
```



```
■    }  
■    }
```

现在我们需要修改 Level1.as 文件来添加 2 个新元素进去.将下列代码添加到 Level1.as 文件中的变量声明部分。

```
enemyShotDelay = 10;
```

```
enemyHealthPoints = 2;
```

## 完成 NoTanks.as 文件

现在我们将要快速浏览所有的代码并尽可能的将 GameDemoInteration6.as 文件变成一个完成的游戏.首先有一些琐碎的事情需要我们来完成,当然最复杂代码我们已经在前面完成(?).当我们完成时,你将拥有一个可以设置更多关卡并支持扩展你的任何创想的完整游戏。

正如更改所说,在此轮更改中,我们将更改并将其保存为一个新类。

下面是该新类在 Flash IDE 中的名字和位置:

```
/source/projects/notanks/flashIDE/com/efg/games/notanks/NoTanks.as
```

下面是该新类在 Flex SDK 中的文字和位置:

```
/source/projects/notanks/flexSDK/src/com/efg/games/notanks]NoTanks.as
```

将其更名为 NoTanks.as



我们的游戏已经有了一个新名字,所以我们需要在修改类名来达到一致.同样我们需要修改构造函数的名字。

将 NoTanks 改为扩展(extend)Game 而非 Sprite 这一点非常重要.下面是新类的定义.首先我们将其改名。

```
public class NoTanks extends Game
```

```
{
```

and then the constructor name:

```
public function NoTanks() {
```

为 NoTanks.as 类添加最终要的部分.

新的 NoTanks.as 的最顶端看起来应该向下面的代码一样;你不一定要将注释也敲进去,但是如果你需要你可以将他们保留用来查阅和参考。

```
■ package com.efg.games.notanks
■ {
■
■     import flash.display.BitmapData;
■
■     import flash.geom.Point;
■     import flash.geom.Rectangle;
■     import flash.display.Bitmap;
■     import flash.display.Sprite;
■     import flash.events.*;
■     import com.efg.framework.CustomEventSound;
■     import com.efg.framework.Game;
■     import com.efg.framework.CustomEventLevelScreenUpdate;
■     import com.efg.framework.CustomEventScoreBoardUpdate;
```



```
■ import com.efg.framework.CustomEventSound;
■ import com.efg.framework.BlitSprite;
■ import com.efg.framework.TileByTileBlitSprite;
■ import com.efg.framework.TileSheet;
■
■
■ /**
■  * ...
■  * @author Jeff Fulton
■  */
■ public class NoTanks extends Game {
```

为新的 NoTanks.as 添加变量

将下面的代码添加到变量定义的部分:

```
■ public static const EXPLODE_SMALL:int = 0;
■ public static const EXPLODE_LARGE:int = 1;
■ //game specific
■ private var score:int;
■ private var lives:int;
■ private var ammo:int;
■ private var playerStartLives:int = 2;
■ private var playerStartAmmo:int = 50;
■ private var playerStartHealthpoints:int = 5;
■ private var scoreEnemy:int = 25;
■ private var scoreGoal:int = 50;
■
■ //game loop
■ private var gameOver:Boolean = false;
■
■ //goal
■ private var goalReached:Boolean = false;
■ private var goalSprite:BlitSprite;
■
■ //level specific
■ private var enemyIntelligence:int;
■ private var enemyShotDelay:int;
■ private var enemyShotSpeed:int;
```



```
■ private var enemyHealthPoints:int;
■
■
■ //hit detection
■ private var playerHitPoint:Point = new Point(0, 0);
■ private var enemyHitPoint:Point = new Point(0, 0);
■ private var missileHitPoint:Point = new Point(0, 0);
■
■ private var pickupHitPoint:Point = new Point(0, 0);
■
■
■ //temps for loops
■ private var tempExplode:BlitSprite;
■ private var tempPickup:BlitSprite;
■
■ //explosions
■ private var explosionList:Array;
■
■ //pickups
■ private var ammoPickupList:Array;
■ private var lifePickupList:Array;
■ private var ammoPickupAmount:int;
■
■ //missiles
■ private var playerMissileList:Array;
■ private var tempMissile:BlitSprite;
■ private var enemyMissileList:Array;
■ /** end the final Game.as variables **
```

## 为新的 NoTanks.as 添加 init 函数

新的 init 函数将移除调用 newGame 函数代码并且调用设置 focusRect 属性为 false。

```
■ private function init():void {
■     this.focusRect = false;
■     initTileSheetData();
■ }
```

现在 newGame 函数由 Main.as 调用并且当新的关卡开始时 focusRect 属性将被设置为 false。

## 创建 newGame 函数



这个函数基本上重写了老的 newGame 函数。我们添加了新代码来重设 level, score, lives 和 GameOver 变量, 以及广播事件来刷新得分板上的四个元素。

```
■ override public function newGame():void {
■     setRegions();
■
■     level = 0;
■     score = 0;
■     lives = playerStartLives;
■     gameOver = false;
■
■     stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
■     stage.addEventListener(KeyboardEvent.KEY_UP, keyUpListener);
■
■     addChild(canvasBitmap);
■     dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
■         UPDATE_TEXT, Main.SCORE_BOARD_SCORE, "0"));
■
■     dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
■         UPDATE_TEXT, Main.SCORE_BOARD_AMMO, "0"));
■
■     dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
■         UPDATE_TEXT, Main.SCORE_BOARD_TANKS, "0"));
■
■     dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
■         UPDATE_TEXT, Main.SCORE_BOARD_HEALTH, "0"));
■ }
```

## 创建 newLevel 函数

newLevel 函数必须将所有涉及新关卡的变量都重置。它还要确定我们是否正在开始一个新的游戏 (level == 0), 如果是我们需要调用 restartPlayer 并传入对应的参数值(true)来保证玩家开始游戏时其 ammo



和 healthPoints 属性被设置为一个新坦克的属性。我们添加并修改了很多行代码,所以最好的方法是全部重写这个函数。

```
■ override public function newLevel():void {  
■  
■     stage.focus = this;  
■     var newGameStart:Boolean;  
■  
■     //if new game then reset ammo and health  
■     if (level == 0) {  
■         newGameStart = true;  
■     }else {  
■         newGameStart = false;  
■     }  
■  
■     if (level==levels.length-1) level = 0;  
■     level++;  
■     player = new TileByTileBlitSprite(tileSheet, playerFrames, 0);  
■     player.missileTime = 0;  
■     player.missileDelay = 2;  
■  
■     enemyList = [];  
■     playerMissileList = [];  
■     explosionList = [];  
■     enemyMissileList = [];  
■     ammoPickupList = [];  
■     lifePickupList = [];  
■     goalReached = false;  
■     readBackGroundData();  
■     readSpriteData();  
■     drawLevelBackGround();  
■     goalReached = false;  
■  
■     dispatchEvent(new  
CustomEventLevelScreenUpdate(CustomEventLevelScreenUpdate.  
■         UPDATE_TEXT, String(level)));  
■  
■     restartPlayer(newGameStart);  
■     addChild(player);
```



```
■    }
```

这个函数还会将 level 值作为一个字符串传回到 Main 中,该传回值会在广播 (dispatching)CustomEventLevelScreenUpdate 类的实例时被用到 levelInScreen 中。还有一点需要注意我们设置了 stage.focus=this 来确保游戏中的所有键盘事件都会被捕获。

## 创建 restartPlayer 函数

restartPlayer 函数仅仅需要增加一点点改动。当游戏是从新开始或者玩家已经死亡时我们需要在 restartPlayer 函数中添加重置 player.healthPoints 值的代码。这样做之后,当开始一个新的关卡时,玩家的生命和弹药就会和上一个关卡中的一致了(当然你也可以按照你意愿来更改这个设置)。我们还需要替换掉创建 ENTER\_FRAME 监听器。它将会被框架中的 timer 取代。确保函数的所有代码都被覆盖:

```
■ private function restartPlayer(afterDeath:Boolean=false):void {
■     trace("restart player");
■     player.visible = true;
■     player.currCol = playerStartCol;
■     player.currRow = playerStartRow;
■     player.x=(playerStartCol * tileWidth)+(.5*tileWidth);
■     player.y = (playerStartRow * tileHeight) + (.5 * tileHeight);
■     player.nextX = player.x;
■     player.nextY = player.y;
■     player.currentDirection = MOVE_UP;
■     playerStarted = true;
■     playerInvincible = true;
■     playerInvincibleCountDown = true;
■     playerInvincibleCount = 0;
■
■     /** added iteration 6
■     setCurrentRegion(player);
■     /** end added iteration 6
■
■     /** added in Game.as final ***
■     player.healthPoints = playerStartHealthpoints;
```



```
■      if (afterDeath) {  
■          ammo = playerStartAmmo;  
■          player.healthPoints = playerStartHealthpoints;  
■      }  
■      /** end added in Game.as final **  
■  }
```

## 重写 runGame 函数

这三个函数是最新加入的函数,你只需要将他们准确的拷贝到 Notanks.as 文件中,接着重写 (overwriting)当前的 runGame 函数即可。

在每一帧,updateScoreBoard 函数都会刷新得分板(scoreBoard),当开始一个新的关卡或 开始一个新坦克时,玩家的坦克在一段时间内不会受到任何伤害,checkInvincible 函数就是被用来计数该时间的.当玩家处于无敌时,我们的一些效果(update 和 render 循环中的一部分效果)会让玩家坦克闪烁。

现在 runGame 函数已经被所有必要的游戏循环函数所填满.它还会检查游戏或是关卡是否结束,然后更新得分板(scoreBoard).直到 Boolean 类型的 playerStarted 被设置为 true 时它才会监听键盘输入的事件。

```
■      private function updateScoreBoard():void {  
■  
■          dispatchEvent(new  
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.  
■              UPDATE_TEXT, Main.SCORE_BOARD_SCORE, String(score));  
■  
■          dispatchEvent(new  
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.  
■              UPDATE_TEXT,Main.SCORE_BOARD_AMMO,String(ammo));  
■      }
```



```

■      dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
■      UPDATE_TEXT,Main.SCORE_BOARD_TANKS,String(lives)));
■
■      dispatchEvent(new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
■      UPDATE_TEXT,Main.SCORE_BOARD_HEALTH,String(player.healthPoints)
■      + "/" + String(playerStartHealthpoints)));
■
■      }
■
■      override public function runGame():void {
■      checkInvincible();
■      if (playerStarted) {
■      checkInput();
■      }
■      update();
■      checkCollisions();
■      render();
■
■      checkforEndLevel();
■      checkforEndGame();
■      updateScoreBoard();
■
■      }
■
■      private function checkInvincible():void {
■      if (playerInvincibleCountDown && playerInvincible) {
■      playerInvincibleCount++
■      if (playerInvincibleCount > playerInvincibleWait) {
■      playerInvincible = false;
■      playerInvincibleCountDown = false;
■      playerInvincibleCount = 0;
■      player.visible = true;
■      }
■      }
■      }
■      }

```

## 添加 CheckInput 函数



我们需要将 checkInput 函数中某些用来播放声音和开火的代码解除注释.如下代码所示:

```
private function checkInput():void {
    var playSound:Boolean = false;
    var lastDirection:int = player.currentDirection;

    if (keyPressList[38] && player.inTunnel==false) {

        if (checkTile(MOVE_UP, player )) {
            if (player.currentDirection == MOVE_UP || player.currentDirection
            == MOVE_DOWN || player.currentDirection == MOVE_STOP ) {

                switchMovement(MOVE_UP, player );
                playSound = true;

            }else if (checkCenterTile( player)) {
                switchMovement(MOVE_UP, player );
                playSound = true;
            }
        }else{
            //trace("can't move up");
        }
    }

    if (keyPressList[40] && player.inTunnel == false) {
        if (checkTile(MOVE_DOWN, player )) {
            if (player.currentDirection == MOVE_DOWN || player.currentDirection
            == MOVE_UP || player.currentDirection == MOVE_STOP) {

                switchMovement(MOVE_DOWN, player );
                playSound = true;

            }else if (checkCenterTile(player)) {

                switchMovement(MOVE_DOWN, player );
                playSound = true;
            }
        }
    }
}
```



```

    }else {
        //trace("can't move down");
    }
}

if (keyPressList[39] && player.inTunnel==false) {
    if (checkTile(MOVE_RIGHT, player )) {
        if (player.currentDirection == MOVE_RIGHT || player.currentDirection
            == MOVE_LEFT || player.currentDirection == MOVE_STOP) {

            switchMovement(MOVE_RIGHT, player );
            playSound = true;

        }else if (checkCenterTile( player)) {

            switchMovement(MOVE_RIGHT, player );
            playSound = true;

        }
    }else {
        //trace("can't move right");
    }
}

if (keyPressList[37] && player.inTunnel==false) {
    if (checkTile(MOVE_LEFT,player )) {
        if (player.currentDirection == MOVE_LEFT || player.currentDirection
            == MOVE_RIGHT || player.currentDirection == MOVE_STOP) {

            switchMovement(MOVE_LEFT, player );
            playSound = true;

        }else if (checkCenterTile(player)) {

            switchMovement(MOVE_LEFT, player );
            playSound = true;

        }
    }else {
        //trace("can't move left");
    }
}
```



```
■    }
■
■
■    if (keyPressList[32]&& player.inTunnel==false) {
■        if (ammo >0) firePlayerMissile();
■    }
■
■    if (lastDirection == MOVE_STOP && playSound) {
■        dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
■        Main.SOUND_PLAYER_MOVE, false, 1, 0));
■    }
■    }
```

下面是我们所做更改的列表:

我们将代表是否播放坦克移动声音的变量所在行解除注释: `var playSound:Boolean = false;`

`if:else` 从句中 应该有 8 个地方需要将代码 `playSound = true` 行解除注释;

在函数的底端,当 空格键被按下 `:if (ammo >0) firePlayerMissile();`所在行解除注释

当第二步中的 `playSound` 被设置为 `true` 时,则需要播放 `SOUND_PLAYER_MOVE` 声音, 将该行代码解除注释.

```
■ if (lastDir == MOVE_STOP && playSound) {
■     dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, É
■     Main.SOUND_PLAYER_MOVE, false, 1, 0));
■ }
```

## 改进 update 函数

我们添加一些代码来更新无敌状态下的玩家,游戏中的子弹和爆炸.要注意的是当一个子弹碰到墙时,它会一起一个小的爆炸,然后被传入 `dispose` 函数来删除.这一过程同样存在于帧动画的过程中。



将下列代码添加到当前 update 函数中

```
■    /** added in the final Game.as **  
■        if (playerInvincibleCountDown) {  
■            if (playerInvincibleCount % 2 == 0) {  
■                //blink player  
■                player.visible = !player.visible  
■            }  
■        }  
■        //player missiles  
■        var playerMissileLength:int = playerMissileList.length - 1;  
■  
■        for (ctr = playerMissileLength; ctr >= 0; ctr--) {  
■            tempMissile = playerMissileList[ctr];  
■            tempMissile.nextX = tempMissile.x + tempMissile.dx ;  
■            tempMissile.nextY = tempMissile.y + tempMissile.dy ;  
■  
■            if (tempMissile.nextY <= yMin-.5*tileHeight ||  
■                tempMissile.nextY >= yMax+.4*tileHeight ||  
■                tempMissile.nextX <= xMin-.5*tileWidth ||  
■                tempMissile.nextX >= xMax+.5*tileWidth) {  
■  
■                playerMissileList.splice(ctr,1);  
■                dispose(tempMissile);  
■            }else if (checkHitWall(tempMissile)) {  
■                playerMissileList.splice(ctr, 1);  
■  
■                createExplode(EXPLODE_SMALL, tempMissile.x, tempMissile.y);  
■  
■                dispatchEvent(new  
CustomEventSound(CustomEventSound.PLAY_SOUND,  
■                    Main.SOUND_HIT_WALL, false, 1, 0));  
■  
■                dispose(tempMissile);  
■            }  
■        }  
■    }
```



```
■  
■  
■    //enemy missiles  
■    var enemyMissileLength:int = enemyMissileList.length - 1;  
■  
■  
■    for (ctr = enemyMissileLength; ctr >= 0; ctr--) {  
■        tempMissile = enemyMissileList[ctr];  
■        tempMissile.nextX = tempMissile.x + tempMissile.dx;  
■        tempMissile.nextY = tempMissile.y + tempMissile.dy;  
■  
■  
■        if (tempMissile.nextY <= yMin-.5*tileHeight ||  
■            tempMissile.nextY >= yMax+.4*tileHeight ||  
■            tempMissile.nextX <= xMin-.5*tileWidth ||  
■            tempMissile.nextX >= xMax+.5*tileWidth) {  
■  
■            enemyMissileList.splice(ctr,1);  
■            dispose(tempMissile);  
■        }else if (checkHitWall(tempMissile)) {  
■            enemyMissileList.splice(ctr, 1);  
■  
■  
■        createExplode(EXPLODE_SMALL, tempMissile.x, tempMissile.y);  
■  
■  
■        dispatchEvent(new  
CustomEventSound(CustomEventSound.PLAY_SOUND,  
■            Main.SOUND_HIT_WALL, false, 1, 0));  
■  
■  
■        dispose(tempMissile);  
■    }  
■  
■  
■  
■  
■  
■  
■  
■  
■    //explosions can be updated and rendered at the same time  
■    var explodeLength:int = explosionList.length - 1;  
■  
■  
■    for (ctr = explodeLength; ctr >= 0; ctr--) {  
■        tempExplode = explosionList[ctr];  
■        tempExplode.animationLoop = true;  
■        tempExplode.updateCurrentTile();  
■  
■  
■        if (tempExplode.loopCounter > 0) {  
■            explosionList.splice(ctr, 1);  
■
```



```
■         dispose(tempExplode);
■     }else {
■         tempExplode.renderCurrentTile();
■     }
■ }
■
■
■     /** end added in the final Game.as
```

## 添加 checkHitWall 函数

checkHitWall 函数是专门被用来检测子弹是否碰撞到墙的。它比 checkTileFunction 函数更简单,只需要一个 TileByTileBlitSprite。因为子弹使用的是 BlitSprite 类并且不需要复杂的移动 AI 逻辑。相对于将子弹硬塞到 tile 的碰撞检测函数 checkTileFunction 中,这个函数将会更快(在代码执行方面)。如果当前子弹 tile 在一个 TILE\_WALL 上它就会返回一个 true。

```
■     private function checkHitWall(object:BlitSprite):Boolean {
■         var row:int = int(object.nextY / tileWidth);
■         var col:int = int(object.nextX / tileHeight);
■
■         return tileSheetData[levelTileMap[row][col]] == TILE_WALL;
■     }
```

## checkCollisions 函数

checkCollisions 函数是一个新的函数; 你从没见过这个函数。在近几章中,你可能看到过类似于为存在于不同的 sprite 中的 BitmapData 实例做完美的像素级碰撞检测的函数。在本章中我们将检测 nextX 和 nextY 的值而非当前的 xy 值。虽然在坦克大战 这类动作游戏中这一检测并非是必要的,但它是一个用来熟悉这一技术的好机会。如果你这样做了,由于碰撞检测,在 update 中更新的物体实例绝对不会移动的过远或者需要往回移动。

下面这些碰撞检测功能就是我们这个函数中需要的功能:



当数组 playerMissiles 中任何一个子弹碰到敌军一个坦克时:敌军坦克也像玩家一样有一定量的生命点数.当失去所有生命点数,敌军坦克就被摧毁了.

数组 playerMissiles 中的一个或多个子弹碰到玩家时:每碰到一个敌军的子弹,就削减一次自己的生命点数.当失去全部的 5 点生命,玩家坦克将爆炸.如果玩家没有生命了,gameOver 变量将被设置为 true.

玩家碰撞到 goalSprite(目标宝藏):一旦玩家碰到 goalSprite(目标宝藏),当前关卡立即结束。

当玩家碰撞到数组 ammoPickupList 中的任意一个 BlitSprite:则变量 ammo 将会伴随 ammoPickupAmount 的值增长而增长。

当玩家碰撞到数组 lifePickups 中的任意一个 BlitSprite 时: 变量 lives 增加 1。

下面是新函数的代码:

```
■ private function checkCollisions():void {  
■  
■     //loop through playerMissiles and check against enemy  
■     var playerMissileLength:int = playerMissileList.length - 1;  
■     playerHitPoint.x = player.nextX;  
■     playerHitPoint.y = player.nextY;  
■  
■     var enemyLength:int = enemyList.length - 1;  
■  
■     missiles: for (var ctr:int = playerMissileLength; ctr >= 0; ctr--) {  
■         tempMissile = playerMissileList[ctr]  
■     }
```



```
■      for (var ctr2:int = enemyLength; ctr2 >= 0; ctr2--) {
■          tempEnemy = enemyList[ctr2];
■          missileHitPoint.x = tempMissile.nextX;
■          missileHitPoint.y = tempMissile.nextY;
■          enemyHitPoint.x = tempEnemy.nextX;
■          enemyHitPoint.y = tempEnemy.nextY;
■
■          if (tempMissile.bitmapData.hitTest(missileHitPoint,
■              255, tempEnemy.bitmapData, enemyHitPoint)) {
■
■              //trace("hit enemy");
■              tempEnemy.healthPoints--;
■              if (tempEnemy.healthPoints < 1) {
■                  createExplode(EXPLODE_LARGE, tempEnemy.x, tempEnemy.y);
■
■                  dispatchEvent(new
CustomEventSound(CustomEventSound.PLAY_SOUND,
■                      Main.SOUND_EXPLODE, false, 1, 0));
■
■                  enemyList.splice(ctr2, 1);
■                  dispose(tempEnemy);
■                  score += scoreEnemy;
■              }else {
■
■                  createExplode(EXPLODE_SMALL, tempMissile.x, tempMissile.y);
■
■                  dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
■                      Main.SOUND_HIT, false, 1, 0));
■              }
■
■              playerMissileList.splice(ctr, 1);
■              dispose(tempMissile);
■
■              break missiles;
■          }
■      }
■
■      //loop through playerMissiles and check against enemy
■      var enemyMissileLength:int = enemyMissileList.length - 1;
```



```
■  
■    emissiles:for (ctr = enemyMissileLength; ctr >= 0; ctr--) {  
■        tempMissile = enemyMissileList[ctr]  
■        missileHitPoint.x = tempMissile.nextX;  
■        missileHitPoint.y = tempMissile.nextY;  
■  
■        if (tempMissile.bitmapData.hitTest(missileHitPoint, 255,  
■            player.bitmapData, playerHitPoint) && !playerInvincible) {  
■  
■            player.healthPoints--;  
■            if (player.healthPoints < 1) {  
■                createExplode(EXPLODE_LARGE, player.x, player.y);  
■  
■                dispatchEvent(new  
CustomEventSound(CustomEventSound.PLAY_SOUND,  
■                    Main.SOUND_PLAYER_EXPLODE, false, 1, 0));  
■  
■                lives--;  
■                if (lives > 0) {  
■                    restartPlayer();  
■                }else {  
■                    gameOver = true;  
■                }  
■  
■                player.visible = false;  
■  
■            }else {  
■                createExplode(EXPLODE_SMALL, tempMissile.x, tempMissile.y)  
■  
■                dispatchEvent(new  
CustomEventSound(CustomEventSound.PLAY_SOUND,  
■                    Main.SOUND_HIT, false, 1, 0));  
■            }  
■  
■            enemyMissileList.splice(ctr, 1);  
■            dispose(tempMissile);  
■            break emissiles;  
■        }  
■    }  
■
```



```

    if (player.hitTestObject(goalSprite)) {
        dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
            Main.SOUND_GOAL, false, 1, 0));

        dispose(goalSprite);
        score += scoreGoal;
        goalReached = true;
        playerInvincible = true;
    }

    var ammoPickupLength:int = ammoPickupList.length - 1;
    for (ctr = ammoPickupLength; ctr >= 0; ctr--) {
        tempPickup = ammoPickupList[ctr];
        pickupHitPoint.x = tempPickup.x;
        pickupHitPoint.y = tempPickup.y;

        if (tempPickup.bitmapData.hitTest(pickupHitPoint, 255, player.bitmapData,
            playerHitPoint)){

            ammoPickupList.splice(ctr, 1);
            dispose(tempPickup);
            ammo += ammoPickupAmount;

            dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
                Main.SOUND_PICK_UP, false, 1, 0));

        }
    }

    var lifePickupLength:int = lifePickupList.length - 1;
    for (ctr = lifePickupLength; ctr >= 0; ctr--) {
        tempPickup = lifePickupList[ctr];
        pickupHitPoint.x = tempPickup.x;
        pickupHitPoint.y = tempPickup.y;

        if (tempPickup.bitmapData.hitTest(pickupHitPoint, 255, player.bitmapData,
            playerHitPoint)){
```



```
■         lifePickupList.splice(ctr, 1);
■         dispose(tempPickup);
■         lives += 1;
■
■         dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
■             Main.SOUND_LIFE, false, 1, 0));
■     }
■ }
■
■ }
```

## 为 NoTanks.as 更新 render 函数

需要在 render 函数中加入一个可以迭代(循环)访问 player 和 enemy 子弹的容器。将下列代码添加到 render 函数中已存在的代码后面,当然要添加在 render 函数结束括号之前。

```
■  /** added in final Game.as
■      var playerMissileLength:int = playerMissileList.length-1;
■
■      for (ctr = playerMissileLength; ctr >= 0; ctr--) {
■          tempMissile = playerMissileList[ctr];
■          tempMissile.x = tempMissile.nextX;
■          tempMissile.y = tempMissile.nextY;
■      }
■
■      var enemyMissileLength:int = enemyMissileList.length - 1;
■      for (ctr = enemyMissileLength; ctr >= 0; ctr--) {
■          tempMissile = enemyMissileList[ctr];
■          tempMissile.x = tempMissile.nextX;
■          tempMissile.y = tempMissile.nextY;
■      }
■
■      /** end added in final Game.as
```

## 检查关卡或游戏结束

我们需要加入 3 个全新的函数,正如下面代码所示。



checkForEndOfLevel 函数检查 goalReached 的 Boolean 值是否为 true。

checkForEndOfGame 函数检查 gameOver 的 Boolean 值是否为 true.它还会检查是否所有的爆炸都结束了.当然这仅仅是一个简单示意的过度例子.你将会看到关卡的结束有些粗糙,但是游戏结束循环更细致一点儿.你可以将你想到的判断条件添加到 2 个函数中来达到你想要的平滑过度效果。

addToScore 函数接收到一个整数,并将它加到 score 上

```
■ private function checkforEndGame():void {  
■  
■     if (gameOver && explosionList.length==0) {  
■         dispatchEvent(new Event(GAME_OVER));  
■         disposeAll();  
■     }  
■ }  
■  
■  
■  
■ private function checkforEndLevel():void {  
■  
■     if (goalReached) {  
■         disposeAll();  
■         dispatchEvent(new Event(NEW_LEVEL));  
■     }  
■ }  
■  
■  
■  
■ private function addToScore(val:Number):void {  
■     score += val;  
■ }
```

## 为发射出的子弹创建函数

firePlayerMissile 和 fireMissileAtPlayer 函数非常相似.事实上通过一些更改可以将他们合并到一个函数中(例如:通过 switch:case 来处理玩家或敌人发射出的子弹的不同状态),但是由于他们还是存在着一些



不同的功能,我们仍然保持他们相互独立.他们两个都用到了 `TileByTileBlitSprite` 类中的 `missileTime` 和 `missileDelay` 属性来判断某个对象是否可以开火.它通过比较 `TileByTileBlitSprite.missileTime` 和基于帧的计数来判断.如果 `missileTime` 比 `missileDelay` 大,将会发射一个子弹.所有的子弹都是 `BlitSprite` 的实例。

一旦子弹被发射出,它的方向将被设置为玩家或者敌军开火时的方向.所有玩家的子弹每帧都会移动 3 像素.如果你想要通过某种方法让玩家子弹移动的速度更快,可以通过通过更改某个变量来设定这个速度(像素每帧)的值.敌人的子弹速度由 `ILevel` 类实例中的 `enemyShotSpeed` 控制。

`firePlayerMissile` 是一个全新的函数.将代码全部复制. `fireMissileAtPlayer` 函数已经存在。

```
■         private function firePlayerMissile():void {  
■         if (player.missileTime++ > player.missileDelay) {  
■             ammo--;  
■             player.missileTime = 0;  
■             //trace("fire a missile");  
■  
■             tempMissile = new BlitSprite(tileSheet, missileTiles, 0);  
■  
■             switch(player.rotation) {  
■  
■                 case 0:  
■                     tempMissile.dx = 0;  
■                     tempMissile.dy = -3;  
■                     tempMissile.x = player.x;  
■                     tempMissile.y = player.y-10;  
■                     break  
■  
■                 case 90:  
■                     tempMissile.x = player.x+10;  
■                     tempMissile.y = player.y;  
■                     tempMissile.dx = 3;  
■                     tempMissile.dy = 0;  
■                     break;  
■  
■                 case 180:
```



```
■          tempMissile.x = player.x;
■          tempMissile.y = player.y+10;
■          tempMissile.dx = 0;
■      tempMissile.dy = 3;
■          break
■
■      case -90:
■          tempMissile.x = player.x-10;
■          tempMissile.y = player.y;
■          tempMissile.dx = -3;
■          tempMissile.dy = 0;
■          break;
■      }
■
■      tempMissile.nextX = tempMissile.x;
■      tempMissile.nextY = tempMissile.y;
■      playerMissileList.push(tempMissile);
■      addChild(tempMissile);
■
■      dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
■          Main.SOUND_PLAYER_FIRE, false, 1, 0));
■      }
■  }
■
■
■      private function fireMissileAtPlayer(enemySprite:TileByTileBlitSprite):void {
■
■          if (enemySprite.missileTime++ > enemySprite.missileDelay) {
■              enemySprite.missileTime = 0;
■              trace("fire a missile");
■              tempMissile =new BlitSprite(tileSheet, missileTiles, 0);
■
■              switch(enemySprite.rotation) {
■
■                  case 0:
■                      tempMissile.dx = 0;
■                      tempMissile.dy = -1*enemyShotSpeed;
■                      tempMissile.x = enemySprite.x;
■                      tempMissile.y = enemySprite.y-10;
■                      break
```



```

case 90:

    tempMissile.x = enemySprite.x+10;
    tempMissile.y = enemySprite.y;
    tempMissile.dx = enemyShotSpeed;
    tempMissile.dy = 0;
    break;

case 180:

    tempMissile.x = enemySprite.x;
    tempMissile.y = enemySprite.y+10;
    tempMissile.dx = 0;
    tempMissile.dy = enemyShotSpeed;
    break

case -90:

    tempMissile.x = enemySprite.x-10;
    tempMissile.y = enemySprite.y;

tempMissile.dx = -1*enemyShotSpeed;
    tempMissile.dy = 0;
    break;
}

tempMissile.nextX = tempMissile.x;
tempMissile.nextY = tempMissile.y;
enemyMissileList.push(tempMissile);
addChild(tempMissile);

dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
    Main.SOUND_ENEMY_FIRE, false, 1, 0));
}
}

```

## 更改 readBackGroundData 函数

一定要确定 Level 类的子类中的 ReadBackGroundData 函数中已经包含了 5 个关卡特殊信息的数据



变量.下面是该函数的例子,代码中粗体部分代表的是 5 个被添加进来的变量.

```
■ private function readBackGroundData():void {  
■     levelTileMap = [];  
■     levelData = levels[level];  
■     levelTileMap = levelData.backGroundMap;  
■  
■     /** The five new variables  
■     enemyIntelligence = levelData.enemyIntelligence;  
■     enemyShotDelay=levelData.enemyShotDelay;  
■     enemyShotSpeed = levelData.enemyShotSpeed;  
■     enemyHealthPoints = levelData.enemyHealthPoints;  
■     ammoPickupAmount=levelData.ammoPickUp;  
■ }
```

## 更改 readSpriteData 函数

readSpriteData 函数需要更新一些处理新 sprite 类型的操作.还有 需要将 enemy sprites 处理操作中的某几行代码解除注释.

首先,将 case SPRITE\_ENEMY 部分中的几行代码解除注释:

```
empEnemy.missileDelay = enemyShotDelay;
```

```
tempEnemy.healthPoints = enemyHealthPoints;
```

下面是 SPRITE\_ENEMY 部分全部代码的参考:

```
■ case SPRITE_ENEMY:  
■     tempEnemy = new TileByTileBlitSprite(tileSheet, enemyFrames, 0);  
■     tempEnemy.x=(colCtr * tileWidth)+(0.5*tileWidth);  
■     tempEnemy.y = (rowCtr * tileHeight) + (0.5 * tileHeight);  
■     tempEnemy.currRow = rowCtr;  
■     tempEnemy.currCol = colCtr;  
■     setCurrentRegion(tempEnemy);  
■     tempEnemy.currentDirection = MOVE_STOP;  
■     tempEnemy.animationLoop = false;
```



```
■ tempEnemy.missileDelay = enemyShotDelay;
■ tempEnemy.healthPoints = enemyHealthPoints;
■ addChild(tempEnemy);
■ enemyList.push(tempEnemy);
■ break;
```

现在 我们将 AMMO, GOAL,和 LIVES sprite 类型的 case 分支:

```
■ case SPRITE_AMMO:
■     tempPickup = new BlitSprite(tileSheet, [ammoFrame], 0);
■     tempPickup.x=(colCtr * tileWidth)+(.5*tileWidth);
■     tempPickup.y = (rowCtr * tileHeight) + (.5 * tileHeight);
■     addChild(tempPickup);
■     ammoPickupList.push(tempPickup);
■     break;
■
■ case SPRITE_GOAL:
■     goalSprite = new BlitSprite(tileSheet, [goalFrame], 0);
■     goalSprite.x=(colCtr * tileWidth)+(.5*tileWidth);
■     goalSprite.y = (rowCtr * tileHeight) + (.5 * tileHeight);
■     addChild(goalSprite);
■     break;
■
■ case SPRITE_LIVES:
■     tempPickup = new BlitSprite(tileSheet, [livesFrame], 0);
■     tempPickup.x=(colCtr * tileWidth)+(.5*tileWidth);
■     tempPickup.y = (rowCtr * tileHeight) + (.5 * tileHeight);
■     addChild(tempPickup);
■     lifePickupList.push(tempPickup);
■     break;
```

不管是玩家或敌人,这三种 sprite 都会用同样的方式来处理.他们的不同之处在于他们是 BlitSprite 类的实例而非 TileByTileBlitSprite 类的实例.他们不需要在迷宫中移动;他们只需要静止保持不动.(这句有些不太确定!!!)。

## 更改 checkLineOfSight 函数



我们需要对 `checkLineOfSight` 函数做一个很小的更改.在前面我们使用了一个占位符来表示 `enemyIntelligence` 值.现在我们可以将占位符删掉了,因为我们可以从 `readBackGroundData` 函数中读到全局变量 `enemyIntelligence`,找到该行代码将其注释掉或彻底删除:

```
/**placeholder
```

```
//var enemyIntelligence:int = 50;
```

## 添加 `createExplode` 函数

`createExplode` 函数用来创建碰墙时的小型爆炸或坦克被摧毁是的爆炸效果。这个函数接收 3 个参数。第一个参数是一个代表了 `EXPLODE_SMALL` 或 `EXPLODE_LARGE` 大小的整形常量。第二个是 x 坐标,最后一个是 y 坐标。爆炸效果会被添加到 `explosionList` 数组中。

```
■ private function createExplode(size:int, xloc:int, yloc:int):void {  
■  
■     if (size==EXPLODE_SMALL) {  
■         tempExplode = new BlitSprite(tileSheet, explodeSmallTiles, 0);  
■     }else {  
■         tempExplode = new BlitSprite(tileSheet, explodeLargeTiles, 0);  
■     }  
■  
■     tempExplode.x = xloc;  
■     tempExplode.y = yloc;  
■     tempExplode.animationLoop = true;  
■     tempExplode.useLoopCounter = true;  
■     explosionList.push(tempExplode);  
■     addChild(tempExplode);  
■ }
```

## 编写清理函数

对象清理函数由 `dispose` 和 `disposeAll` 组成 `dispose` 函数会将接收到的 `BlitSprite`(或者其孩子



TileByTileBlitSprite)从屏幕上一处并并将其置为等待垃圾回收的状态.它会调用 BlitSprite 类内部的 dispose 函数并将所有对象从显示列表中移除。

disposeAll 函数会调用游戏用所有对象的 dispose 函数.它会在每个关卡结束和游戏结束时被调用.

```
■ private function dispose(object:BlitSprite):void {
■     object.dispose();
■     removeChild(object);
■ }
■
■ private function disposeAll():void {
■     dispose(player);
■
■     for each(tempEnemy in enemyList) {
■         dispose(tempEnemy)
■     }
■     enemyList = null;
■
■     for each(tempMissile in playerMissileList) {
■         dispose(tempMissile)
■     }
■     playerMissileList = null;
■
■     for each(tempMissile in enemyMissileList) {
■         dispose(tempMissile)
■     }
■     enemyMissileList = null;
■
■     for each(tempExplode in explosionList) {
■         dispose(tempExplode)
■     }
■     explosionList = null;
■
■     for each (tempPickup in ammoPickupList) {
■         dispose(tempPickup);
■     }
■     ammoPickupList = null;
```



```
■  
■      for each (tempPickup in lifePickupList) {  
■          dispose(tempPickup);  
■      }  
■      lifePickupList = null;  
■      }
```

## 测试最终的游戏

现在你已经有了一个可以玩一关的游戏了。在你的 IDE 选项中,选择 构建(Build), 发布(Publish )或者 测试( Test Movie)来看看我们成就吧。你最先应该看到的是由蓝色 title 组成的菜单并且背景音乐开始播放。点击跳过这部分菜单来开始游戏。

一但游戏开始,你应该会看到如图 7-8 所示的屏幕图像。使用方向键在迷宫中移动,空格键来开火。



图 7-8 游戏最终画面

## 扩展游戏

在这两章中我们已经接触了很多东西,并且快速的将其完成。正如在最终的游戏所示, 你可能会期望再



润色一下游戏。在后面的章节我们实现这些功能,例如添加一个暂停和静音的按钮。我们还会重新创建一个更细致的关卡转换效果,而非这个游戏中粗糙的转场效果。

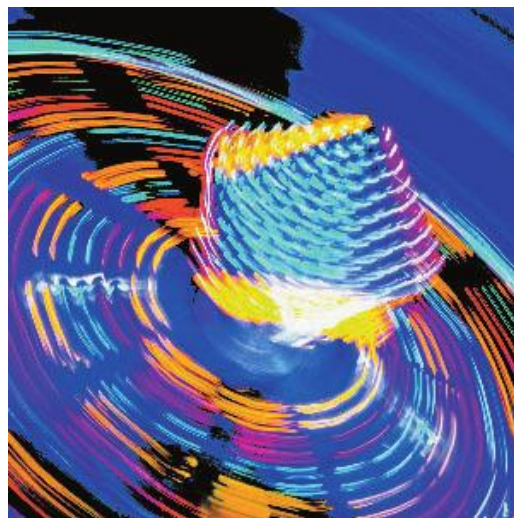
现在你可以做的一件事就是创建更多的关卡。如果你需要复习一下请返回到第六章。如果你添加了更多的关卡,你需要确定他们都是用正确的方法创建的:

- 在 Mappy 中 创建一个 2 图层的关卡
- 将 2 个图层导出存储为 ActionScript
- 创建一个 Level.as 的子类 例如 Level2.as.
- 设置新关卡中的关卡特别信息;例如 你可能会更新 enemyIntelligence 属性
- 将从 Mappy 中导出的文件中读取相关数据,并添加到 backGroundMap 和 spriteMap 中
- 在关卡列表中添加 Level2 的实例引用
- `private var levels:Array = [undefined,new Level1(), new Level2()]`
- 重复这些步骤完成所有的关卡创建

## 关于 Color Drop 游戏

如果你顺利完成了这两章,你可以为自己鼓掌了。当我们接触基础的 tile sheet 时,你是否还记得前面的东西?至此我们通过使用 GIMP 和 Mappy 完成了一个可扩展的游戏关卡创建系统。我们还编写并创建了一个处理多关卡和关卡数据的容器,探索出怎样在基于 tile 的迷宫中让游戏角色移动,怎么样发射子弹,怎么样去编写适用于迷宫游戏的敌军追捕 AI 等等。

在下一章中,我们会在街机类型的游戏上少做停留并使用以前的框架和很多曾经创建好的类来制作一个休闲类游戏。



## 第八章

# 休闲智力游戏--魔法色块

虽然游戏开发者需要确保他们制作的游戏没有用到其他人不允许使用的图形、声音或者文字，但是对于获取游戏创意的行为，这样的规则就没有什么意义了，所以在 下一个章节我们会简短的浏览下关于使用他人创作素材的合法性。作为一名游戏设计者，良好的规范和行为准则有助于您快速的掌握游戏创意的核心内容，并将其融入到自己的游戏中。当然，这并不是意味着你就不能从其他优秀游戏中获取灵感或者使用它的创意来作为您游戏中的基本素材。游戏的创意并不是孤立的，它具备一定的连续性，并与其他事物保持一定的关联。想要了解休闲游戏的发展历程没有比这更适合的例子了。

在这一章，我们将开始制作两个游戏，用两个章节的内容来讨论一下关于现代休闲游戏的风格，并且制作一个名为“魔法方块”的简易休闲拼图类游戏。在下一章节，在我们开始进入制作骰子类的休闲游戏之前，我们将通过谈论一些游戏中的涉及到的智力属性来完成我们关于休闲游戏的讨论。

## 了解休闲游戏的发展历程

大多数原始的电视游戏要追溯 20 世纪 70 年代，像是“碰碰弹子台”和“打砖块”游戏，这两种类型的游戏就是今天休闲游戏的典型代表。但在那时，它们几乎是没有什么区别的电视游戏。在那时，相对于数量极少的游戏来说，它们就真的没必要被严格的划分游戏类型了。事实上，在那时，几乎每一款新的游



戏都能自成一派。与此同时，当时大部分的游戏都被设计成了由 2 个玩家对战的形式。这些游戏的这种竞赛风格的设计，如美式撞球，投标游戏和弹球游戏全部都继承了他们在酒吧或者街道上的玩法。然而与此同时，1978 年，在某些地方，太空入侵的主题被带入到他们在街道所玩儿的游戏中，休闲游戏开始发生了变化。单人游戏、高得分的玩法开始形成，这种玩法强调玩家要打败机器要远胜于打败其他玩家。这种趋势被一款叫做“Tempest and Defender”的游戏持续引领着，它第一次建立起一种可以达到高级别的高难度和高强度的游戏，这种游戏就被作为核心的游戏在一天内被广大的玩家所知晓。这种单人控制的竞争性街道游戏也就被移植到了家庭电视游戏上。到 1982 年的时候，大部分的家庭电视游戏系统的设计都是移植街道游戏或者建立在街道游戏的基础上的。第一次电视休闲游戏玩法被这种技术流的玩法如此闪电般迅速地改变了。从某种角度来看，它具有争议性的问题在于，由于它过分强调单人游戏中以技术为基础的街道竞赛，就等于是创建了一个狭小的盒子来引导玩家从不关心也从不帮助电视游戏的设计者推动电视游戏的发展（然而，这是一种极端单纯的观点，事实是还有许多其他的因素混杂在里面）。

1986 年，当任天堂在美国高速发展时，它开发了一种新的竞赛游戏类型叫做核心游戏，它继承和发展了简单的街道竞赛。（“核心游戏”是一个时期，现在通常被用来描述那些通过潜心研究游戏中高级别和高难度的竞赛而使得游戏得以繁荣发展的游戏发烧友）。任天堂开发的游戏有“超级马里奥兄弟”和“塞尔达传说”都是建立在严谨的早期游戏像是雅达利主机上的“玛雅人的冒险”，并且这些游戏提出了更多的强迫性的和独占性的游戏玩法的经验，而且这些玩法超越了当时普遍单屏电视游戏黄金时期的玩法（大概在 1978-1983），也使得电视游戏工业开始重获新生。具有讽刺意味的是，现在，这些核心游戏仍然被视为休闲游戏，但是这就是连续的游戏设计。

休闲游戏一直是一种无法定义的游戏类，直到 Alexey Pajitnov 在二十世纪八十年代开发的俄罗斯方块游戏从苏联扩散到西方世界。当时，在休闲谜题游戏类型上还有许多其他的例子（例如：在雅达利主机上的夺旗游戏），直到俄罗斯方块游戏的出现他们才进入主流的游戏市场。俄罗斯方块游戏使用了四种经典的形状，每种形状有四个方块。当每个方块落到屏幕的底部时，玩家必须要控制这些方块，把它们摆放到合适的位置上。当方块填满了一整行之后，这一整行的方块将被移除。玩家会因为他们将图形摆放到合适的地方而使方块消除而感到高兴。

相对于被称为拼图游戏而言，俄罗斯方块这种游戏其实更应该被称为动作游戏，只是它更多的融入了拼图益智类的元素在里面。俄罗斯方块是一款非常伟大的游戏，它是一种无结局的游戏设计类型。现在，许多其他的游戏都建立在俄罗斯方块的游戏创意上，但是谁也没能拥有它那种简明的伟大设计。今天，即使是现存最好的俄罗斯方块游戏，也在原始的设计上做了轻微的改动。在二十世纪八十年代晚期，俄罗斯方块游戏是销量最好的游戏之一，而且它还被广泛的移植到各种游戏主机上。1991 年，当任天堂的 Game Boy 游戏主机在美国发售时（1989 年开始在日本发售），俄罗斯方块游戏便携版的流行度也随之暴涨，这也证明了俄罗斯方块游戏是居家旅行必备之游戏。

如此，俄罗斯方块的游戏类型由二十世纪八十年代早期的动作类型转变成了益智的休闲类游戏。这样的游戏有 1989 年 Jay Geertsen 开发的宝石方块游戏，它在世嘉游戏主机上很流行。宝石方块的游戏设计是一条直线上的宝石颜色必须一致，三块宝石为一行，则这三块宝石就可以被消除掉。宝石方块游戏方块落下的设计理念来自于俄罗斯方块，但它又改变了消除规则，使得它看起来像是一款新的游戏。

虽然电视游戏一直在发展自己的休闲游戏，但电脑用户已经玩了很多年。同样的，虽然在这之前有许多休闲游戏的例子，但最具优势的休闲游戏当属 1990 年 Windows 3.0 操作系统自带的纸牌游戏。这款免费的游戏在二十世纪九十年代对计算机专业人员，秘书和家庭电脑用户来说可能是用于消遣的最好的游



戏。它可以很快的就开始运行,拥有简易的玩法,而且玩 家可以在很短的时间内完成游戏。当 Windows 95 操作系统发布以后,随系统附带的另一款游戏“扫雷”对休闲类游戏来说是一次长足的发展。扫雷游戏的设计实际上建立在电脑系统上,这些电脑系统都可以追 溯到很多年前。例如:接龙,它就很容易使人上瘾。扫雷这款游戏设定了一个由许多方块组成的网格,这些网格由玩家点击。点击一个方块将显示它周围埋藏雷数量 的信息。游戏的目的在于使玩家标记尽可能多的埋藏的雷,引爆了埋藏的雷也就意味着游戏结束了。扫雷这个游戏的创意在于使用鼠标来选择网格中的方块,虽然不 是创举,但却成为了休闲游戏的主流,并且它的游戏方式作为一种接口使休闲游戏得到了进一步的发展。

在世纪之交的时候,宝开公司的游戏开发者设计了一款革命性的新游戏叫做“宝石迷情”,它的创意融合了宝石方块和扫雷的设计。在这款游戏中,玩家拥有一个由多 彩宝石组成的游戏舞台,它们一个接一个的排列成一个队列。玩家的任务就是在某一时刻交换两个宝石,保证三个宝石在一排上。当三个宝石的颜色匹配后,他们就会一起被消除掉,然后更多的宝石就会从屏幕上方掉到它们消除的位置。虽然是匹配三个宝石,但它的创意仍旧来自于宝石方块,动作元素被移除掉,使得游戏看起 来更像是扫雷游戏的超级加强版。宝开公司不久后就将这款游戏重新命名为“宝石迷阵”,由此一个突破性的游戏类型就此诞生了。

这种类似**对对碰**的游戏类型已经发展并形成了一套自己的游戏规则和派系,且将其设计理念扩展到了角色扮演类游戏中,并形成了许多超越这种简单的对对碰形式的其他的表现形式。

虽然,各种形式的休闲游戏层出不穷,但这些游戏一般都有一些共性的特征存在着:

- 物体出现在类似于网格(落到里面)的形式中。
- 物体以某种方式落到屏幕中,无论是在自己的界面或是用户交互界面。
- 在屏幕中匹配物体(不一定是三个)成了游戏的核心玩法。
- 在很多实例中,物体被鼠标控制着。
- 时间限制通常不被应用到这种游戏类型中,因为它使得玩家思考极为混乱。
- 空间消除的处理过程成为了共有特征。

所以说,休闲游戏一直在发展,并且现在他已经发展成为了一个独立的游戏类型。休闲游戏出现了更多的种类,并不仅仅只局限于拼图类,甚至于每一个派系分支的发 展历程完整讲述的话都可以写一本书出来。事实上,刚才我们提到的很多网格式的休闲游戏被认为可以追踪他们家族的历史。我们相信设计一款休闲游戏没有什么比 看到它成功并且愿意尝试将自己的思维和想法注入到创意中更好的事情了,所以在这一章节,我们将创建这样的游戏——魔法方块——这是一个基于拼图的休闲类小 游戏。

## 设计游戏

对于魔法方块这个游戏来说,我们将从类似对对碰游戏的创意来创建我们自己风格的游戏,该对象就是尽可能的创建一个最大化的多彩箱子链。下面的就是魔法方块游戏概念设计的分解,图 8-1 是游戏的屏幕截图:

- 游戏名称:魔法色块
- 游戏类型:单击配对消除游戏

- 游戏描述：玩家面对的是一个彩色的方块网格。当玩家单击方块时，所有的临近该方块的同色方块将被一起消除。
- 玩家目标：在游戏结束前，尽可能多的使方块消失并转换为得分。
- 敌人描述：无
- 敌人目标：无
- 升级限制：当得分板显示的分数达到当前级别分数要求时玩家升级。
- 游戏结束条件：当玩家没有更多的游戏机会并且没有达到升级所需的分数时游戏结束。
- 难度斜率：难度使用 Level 来区分难度等级。
- 奖励条件：玩家将获得额外的点数（这部分将计入总分），这些点数的多少取决于在一次游戏机会中玩家消除屏幕同色方块的数量。

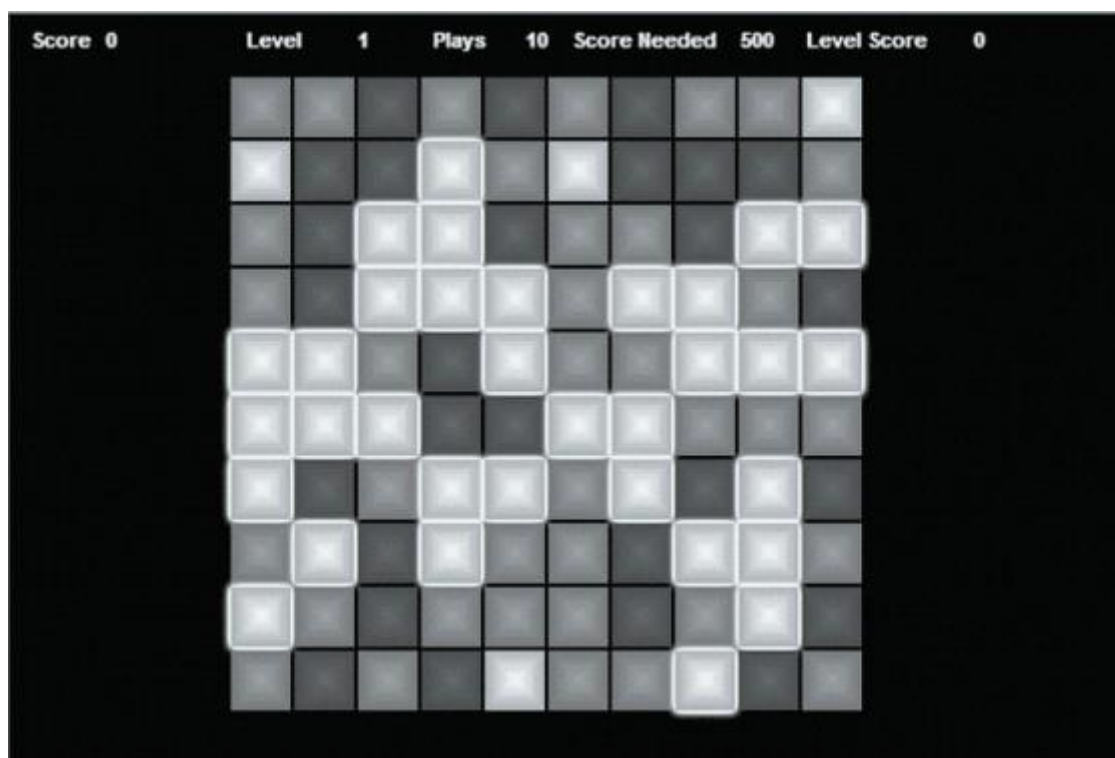


图 8-1. 魔法色块游戏截图

## 这一章的游戏开发概念

下面的是这一章游戏将要涵盖的游戏概念：

- 创建，维护和管理一组网格物体，定义网格的尺寸必须能够通过代码简易的重定义。
- 使用二维数组来有效管理网格的拼图游戏。
- 定义一个有效的迭代循环使得不必递归测试连接。



- 定义一个独特的等级斜率对象来使得等级可以被单独控制。
- 为魔法色块创建一个特殊的状态机来协助游戏业务流的管理。

我们用与这本书其他游戏相似的方式来开始这个项目。你可以使用 Flash IDE 或者使用 Flex/Flash Develop 来创建这个游戏，通常，在必要的情况下我们会记录下使用两种方式创建游戏不同之处。创建一个具有以下属性的新项目（或者 FLA 文件）：

- 包：com.efg.games.colordrop
- 游戏类：ColorDrop.as
- 尺寸：600\*400
- FPS：40
- Flex 素材库定位：src/com/efg/games/colordrop/assets

## 把游戏素材添加到库中

这款游戏不需要太多的素材资源。我们会把这些素材用开发上一章游戏的方法一样将素材嵌入。Flash 素材将被保存在 FLA 文件的库里。Flex 素材将被保存在这个项目的 assets.swf 文件中。

## 添加声音素材

这个游戏只需要少量的声音素材。我们不打算也不打算对 SoundManager 做出什么重大的改变，所以我们将简单的罗列出这个游戏需要用到的声音文件列表。所有这些声音文件都是使用 SFXR（一款游戏音效制作小软件）这个软件制作的：

- 单击音效：当往哪加单击某个方块时播放
- 奖励音效：当一些方块从屏幕消失时播放
- 失败音效：当玩家失败，游戏结束时播放
- 胜利音效：当玩家升级时播放

## 添加图形素材

我们游戏需要的图形素材只有一张尺寸很小的 sprite sheet，它非常的简单。这张图片包括 7 张 32\*32 像素多彩的方块，我们将会使用代码来使它们分离并创建玩家单击的方块对象。如图 8-2 所示。我们必须要把这个制作的更简易，以便一个方块可以使用代码重新上色，并且能够使用代码很好的管理其他颜色。当我们编写程序到达细节阶段时，作为一名程序员，使用何种方法就真的完全取决于你使用哪种方法可以更好的组织和管理素材了。





图 8-2. 魔法色块的 Tile Sheet

## 定义开发魔法色块游戏需要用到的类

在我们进入到编码环节之前，让我们简单地讨论一下对于魔法色块这款游戏我们需要定义哪些类：

- ColorDrop 类：这个是游戏魔法色块的主体类。它会被文档类调用并继承于 com.efg.framework.Game 类。
- Block 类：这个类将描述在屏幕呈现出来的方块。这些方块使用全部使用相同的 tile sheet 素材来呈现在屏幕上。
- CustomEventClickBlock 类：这是一个自定义的事件类，当一个方块被鼠标点击时我们使用这个类的事件通知魔法色块
- GameStates：这个类装载着所有的用来描述游戏状态机的静态变量。

## 更新魔法色块主体类

现在，我们先来看看魔法色块的主体类。这个类看起来你应该很熟悉才是。它继承了 GameFramework 类，并且拥有与其他游戏代码相比独特的代码。代码部分主要的改变在于计分板元件和声音我们将会在游戏中使用到。计分板将会显示下面这些项目：

- Main.SCORE\_BOARD\_SCORE：玩家获得的总分
- Main.SCORE\_BOARD\_LEVEL：当前游戏等级
- Main.SCORE\_BOARD\_PLAYS：玩家还剩下的点击次数
- Main.SCORE\_BOARD\_THRESHOLD：玩家完成当前等级必须要达到的分数
- Main.SCORE\_BOARD\_LEVEL\_SCORE：当前级别下玩家累计的分数

我们之前已经简单地讨论过了游戏中将要使用到的音效，再次，我们定义它们的类分别为 SoundWin，SoundLose，SoundClick 和 SoundBonus，并且它们将被分别引用为 Main.SOUND\_WIN，Main.SOUND\_LOSE，Main.SOUND\_CLICK 和 Main.SOUND\_BONUS。

下面是 Main.as 文件的全部代码；其他的改变（例如：屏幕和标题正文）已被高亮加粗显示：

```
package com.efg.games.colordrop
{
    import com.efg.framework.FrameWorkStates;
    import com.efg.framework.GameFrameWork;
    import com.efg.framework.BasicScreen;
    import com.efg.framework.ScoreBoard;
    import com.efg.framework.Game;
```



```
import com.efg.framework.SideBySideScoreElement;
import com.efg.framework.SoundManager;

import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.display.Sprite;
import flash.events.Event;
import flash.geom.Point;
import flash.utils.Timer;
import flash.events.TimerEvent;
import flash.text.TextFormat;

public class Main extends GameFrameWork {

    public static const SCORE_BOARD_SCORE:String = "score";
    public static const SCORE_BOARD_LEVEL:String = "level";
    public static const SCORE_BOARD_PLAYS:String = "plays";
    public static const SCORE_BOARD_THRESHOLD:String = "threshold";
    public static const SCORE_BOARD_LEVEL_SCORE:String = "levelScore";
    public static const SOUND_CLICK:String = "soundClick";
    public static const SOUND_BONUS:String = "soundBonus";
    public static const SOUND_WIN:String = "soundWin";
    public static const SOUND_LOSE:String = "soundLose";

    /**仅针对 Flex 框架的设定
    */
    [Embed(source = "assets/colordropassets.swf", symbol="SoundClick")]
    private var SoundClick:Class;

    [Embed(source = "assets/colordropassets.swf", symbol="SoundBonus")]
    private var SoundBonus:Class;

    [Embed(source = "assets/colordropassets.swf", symbol="SoundWin")]
    private var SoundWin:Class;
```



---

```
[Embed(source = "assets/colordropassets.swf", symbol="SoundLose")]  
  
private var SoundLose:Class;  
*/  
  
public function Main() {  
    init();  
}  
  
override public function init():void {  
    game = new ColorDrop(600,400);  
    setApplicationBackGround(600,400,false, 0x000000);  
    scoreBoard = new ScoreBoard();  
    addChild(scoreBoard);  
    scoreBoardTextFormat= new TextFormat("_sans", "11", "0xffffffff", "true");  
  
    scoreBoard.createTextElement(SCORE_BOARD_SCORE,  
        new SideBySideScoreElement(10, 5, 15, "Score",  
            scoreBoardTextFormat, 50, "0", scoreBoardTextFormat));  
  
    scoreBoard.createTextElement(SCORE_BOARD_LEVEL,  
        new SideBySideScoreElement(125, 5, 15, "Level",  
            scoreBoardTextFormat, 40, "0", scoreBoardTextFormat));  
  
    scoreBoard.createTextElement(SCORE_BOARD_PLAYS,  
        new SideBySideScoreElement(225, 5, 15, "Plays",  
            scoreBoardTextFormat, 40, "0", scoreBoardTextFormat));  
  
    scoreBoard.createTextElement(SCORE_BOARD_THRESHOLD,  
        new SideBySideScoreElement(300, 5, 15, "Threshold",  
            scoreBoardTextFormat, 80, "0", scoreBoardTextFormat));  
  
    scoreBoard.createTextElement(SCORE_BOARD_LEVEL_SCORE,  
        new SideBySideScoreElement(425, 5, 15, "Level Score",  
            scoreBoardTextFormat, 80, "0", scoreBoardTextFormat));
```



```
screenTextFormat = new TextFormat("_sans", "14", "0xffffffff", "true");
screenButtonFormat = new TextFormat("_sans", "11", "0x000000", "true");
titleScreen = new BasicScreen(
    FrameWorkStates.STATE_SYSTEM_TITLE, THE ESSENTIAL GUIDE TO FLASH
GAMES
    600,400, false, 0x000000);
titleScreen.createDisplayText("Color Drop",
    250, new Point(255, 100), screenTextFormat);
titleScreen.createOkButton("Go!",new Point(250,250),100,20,
    screenButtonFormat,0xFFFFFFFF,0x00FF0000,2);

instructionsScreen = new BasicScreen(
    FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS, 600,400, false, 0x000000);
instructionsScreen.createDisplayText("Click Colored Blocks",300,
    new Point(210,100),screenTextFormat);
instructionsScreen.createOkButton("Play",new Point(250,250),
    100, 20, screenButtonFormat, 0xFFFFFFFF, 0xFF0000, 2);
gameOverScreen = new BasicScreen(
    FrameWorkStates.STATE_SYSTEM_GAME_OVER,
    600, 400, false, 0x000000);
gameOverScreen.createDisplayText("Game Over",300,
    new Point(250, 100), screenTextFormat);
gameOverScreen.createOkButton("Play Again",
    new Point(225, 250), 150, 20,
    screenButtonFormat,0xFFFFFFFF,0xFF0000,2);
levelInText = "Level ";
levelInScreen = new BasicScreen(
    FrameWorkStates.STATE_SYSTEM_GAME_OVER,
    600,400, false, 0x000000);
levelInScreen.createDisplayText(levelInText,300,new Point(275,100),
    screenTextFormat);

switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
waitTime= 40;
```



```
        soundManager.addSound(SOUND_CLICK,new SoundClick());
        soundManager.addSound(SOUND_BONUS, new SoundBonus());
        soundManager.addSound(SOUND_WIN,new SoundWin());
        soundManager.addSound(SOUND_LOSE,new SoundLose());

        frameRate = 40;
        startTimer();
    }

}

}
```

## 创建方块类

我们还需要为魔法色块这个游戏创建一些额外的类。首先是方块类。因为方块是相当复杂的类，把它从系统中优先分离出来这是个好主意。因为大多数魔法色块游戏素材用的都是彩色的方块，所以这个类对于我们的游戏也是最重要的类。

首先 我们创建这个类并且引入需要的类。让这个类继承我们在第 7 章所创建的游戏框架中的 BlitSprite 类。BlitSprite 是一个继承自 Sprite 的类，但是它特殊的地方在于它既用于显示图片素材也用于使用图片素材动起来。

```
package com.efg.games.colordrop
{

    import flash.filters.GlowFilter;
    import flash.events.MouseEvent;
    import flash.display.Bitmap;
    import flash.display.BitmapData;

    import com.efg.framework.TileSheet;
    import com.efg.framework.BlitSprite;

    public class Block extends BlitSprite {
```



现在我们将创建方块类的实例属性。blockColor 属性表示方块的颜色。它的值是马上列出来的 Block.BLOCK\_COLOR\_XXX 的静态类属性 中的一个。下一步，我们设置两个布尔值来表示方块目前正在操作中。它们是方块的缩微状态，但是他们也简单到还不至于使用状态机的设计。这两个缩微状态是 isFalling（表示方块正在朝屏幕下端移动中）和 isFading（方块正在淡出状态中）。对于方块的渐变，我们设定一个名为 fadeValue 的 变量，使用它来使方块在每一帧都产生一定数值的渐变。对于方块的下降状态，fallEndY 变量时方块在 isFalling 变量设置成 false 之前必须 要达到的屏幕 y 轴坐标值。最终，我们设定 speed 属性来定义方块下降移动的速度；设定 nextYLocation 属性来刷新和渲染屏幕显示，就像我们书中之前提到的游戏一样。

```
public var blockColor:Number;
public var isFalling:Boolean;
public var isFading:Boolean;
public var fadeValue:Number = .05;
public var fallEndY:Number;
public var speed:int = 10;
public var nextYLocation:int = 0;
```

另外两个非常有用的实例属性是 row 和 col。这两个属性标识方块在网格中的行与列的位置信息。当一个相同颜色的方块被点击，这两个属性将直接决定这个方块是否需要被检测。

```
public var row:Number;
public var col:Number;
```

方块类中剩下的属性全部都是静态变量，它们用来表示在 tile sheet 中的颜色（见图 8-2）。这些数值将被用于计算 tile sheet 的哪部分将被切割成方块。

```
public static const BLOCK_COLOR_RED:int      = 0;
public static const BLOCK_COLOR_GREEN:int    = 1;
public static const BLOCK_COLOR_BLUE:int     = 2;
public static const BLOCK_COLOR_VIOLET:int   = 3;
public static const BLOCK_COLOR_ORANGE:int   = 4;
public static const BLOCK_COLOR_YELLOW:int   = 5;
public static const BLOCK_COLOR_PURPLE:int    = 6;
```

方块类的构造函数需要从它的父类和自身得到一些参数。由之前的描述可知，blockColor，row，col 和 speed 这些参数是类自身声明的属性。endY 将被传递给 startFalling 属性，并依次将 fallEndY 属性设置成 endY 属性的值。tileSheet 是一个 TileSheet 对 象的实例，放调用父类的一些属性或者方法时，这个属性是必不可少的。这个属性我们会在后面用到它时继续讨论。

TileSheet 会在游戏中创建并传递给方块对象。方块对象并不会创建自己的 TileSheet 对象，这意味着我们不需要为每一个方块对象创建多类型的 TileSheet 对象，这么做只会更多的占用内存。如果在两种方法中选择一种的话，我们可以将 tileSheet 定义成静态属性。然而，我们认为通过传值的来给 tileSheet 赋



值的方式将会使方块类更具通用性和重用。这么做也可以避免我们必须用代码方式来确定 tileSheet 对象是否已经存在和确定 tileSheet 对象作为类属性是否需要被初始化。

接下来的事情就是决定图片素材中的块表示方块的颜色。颜色值将被存储在 blockColor 属性中，而且 Block.BLOCK\_COLOR\_XXX 静态常量表示 tile sheet 中颜色的排列顺序。我们可以非常容易地计算出哪一个块表示哪个方块，就像这样：tile = blockColor;

最后，我们使用代码 super(tileSheet, [tile], 0)调用 BiltSprite 类的构造方法。第一个参数是我们之前提到过的 TileSheet 对象。第二个参数是实例在动画序列中的索引值数组。既然我们的方块不会出现动画，那么我们只需要吧一个简单的值（tile 属性）传过去就可以了。第三个参数是块在块序列数组总的索引值。因为我们只有一个块，所以这个索引值就是 0。

当播放器点击它的时候，我们需要方块派发一个鼠标事件，所以我们要在方块类中添加一个鼠标事件监听 onMouseDownListener。同样的，buttonMode 属性和 useHandCursor 属性都将设置为 true，这样当鼠标指针移动到方块上面时鼠标指针将变成一个引导用户点击的样式。最后我们调用 startFalling 方法来使方块在屏幕中向下移动。

```
public function Block(color:Number,tileSheet:TileSheet,row:Number,col:Number,
endY:Number,speed:Number) {
    blockColor = color;
    this.row = row;
    this.col = col;
    isFalling = false;
    isFading = false;
    var tile:Number = blockColor;
    super(tileSheet, [tile], 0);
    this.addEventListener(MouseEvent.CLICK, onMouseDownListener, false, 0, true);
    this.buttonMode = true;
    this.useHandCursor = true;
    startFalling(endY,speed);
}
```

startFalling 方法是关系到这款游戏方块下落的首要的方法。这个函数的核心在于速度和 fallEndY 实例属性的关系。方块（游戏中设定的）的起始 y 轴坐标和 fallEndY 之间的距离应该可以被速度整除，但事实上没那个必要。如果那么做的话，方块可以非常平滑的落到下面，但是这不会对游戏玩法有任何影响。当这些方法被调用时，我们设置 isFalling 属性为 true 就可以刷新并渲染屏幕，也使得这些方法可以处理方块的移动。

```
public function startFalling(endY:Number,speed:Number):void {
    this.speed = speed;
    fallEndY = endY;
```



```
isFalling = true;  
}
```

startFade 方法的作用是当当前级别通过后使方块从屏幕中淡出。这个淡出的过程通过设置 isFading 属性为 true 而停止。fadeValue 是一个用来表示方块的 alpha 值减少量的参数，当游戏开始渲染时。

```
public function startFade(fadeValue:Number):void {  
    this.fadeValue = fadeValue;  
    isFading=true;  
}
```

这个游戏的刷新方法非常像我们之前已经写过的其他游戏的刷新方法。它看起来是这样的，如果方块的 isFalling 变量等于 true，那么它将会将 nextYLocation 的值更新为方块当前 y 轴坐标加上 speed 变量的值。方块的淡出效果不会影响到渲染的操作。

```
public function update():void {  
    if (isFalling) {  
        nextYLocation = y + speed;  
    }  
}
```

渲染方法只是比刷新方法稍微的复杂了一点儿。如果方块正在下落过程中，那么刷新方法起到的作用就是将方块的 y 轴坐标设置成 nextYLocation 的值。如果方块的 y 轴坐标等于 fallEndY 的值，那么这个方法将通过设置 y 轴坐标等于 fallEndY 的值来使方块最终落到准确的位置。以上我们所做的这些 是因为，方块最后一次移动时，方块的 y 轴坐标值很有可能会超过 fallEndY 的值，所以我们只能简单地让它落到它该去的位置。isFalling 属性被 设置成 false，这样游戏框架才能测试判断出方块是否已经停止移动。

如果方块淡出，我们通过减去 fadeValue 的值来达到更新方块 alpha 属性值的目的。如果方块的 alpha 值小于等于 0 时，我们将 alpha 的值设置 为 0，并且将 isFading 属性设置为 false，这样游戏框架才能测试判断出是否所有淡出的方块已经完成了淡出的动画效果。

```
public function render():void {  
  
    if (isFalling) {  
        y = nextYLocation;  
        if (y >= fallEndY) {  
            y = fallEndY;  
            isFalling = false;  
        }  
    }  
}
```

```
    if (isFading) {
```



```
alpha -= fadeValue;
if (alpha <= 0) {
    alpha = 0;
    isFading = false;
}
}
}
```

当玩家点击使用鼠标点击方块时，onMouseDownListener 方法将被调用。这个方法派发了一个我们下一章才创建的自定义事件。这个自定义事件将自身的引用作为第二个参数传递给监听的方法。我们的游戏将会监听者这个事件，所以它才能开始处理哪一个方块应该从屏幕上被清除。

```
public function onMouseDownListener(e:MouseEvent):void {
    dispatchEvent(new CustomEventClickBlock(CustomEventClickBlock.EVENT_CLICK_BLOCK,
    this));
}
```

makeBlockClicked 方法给方块生成一个发光滤镜。当那个类想通过玩家点击方块的操作来确定哪个方块将发光时这个方法将被调用。

```
public function makeBlockClicked():void {
    this.filters=[new GlowFilter(0xFFFFFFFF,70,4,4,3,3,false,false)]
}
}
}
```

## 创建 CustomEventBlock 类

继续我们的 Block 类，当玩家点击方块时，我们需要创建一个新的自定义事件。这个自定义事件类似于我们在第二章放入游戏框架的自定义事件。不管怎么样，我们需要把这个即将创建的新的自定义事件放入到游戏框架中替换之前的自定义事件。这是为什么呢？因为它对我们的游戏来说足够特殊，以至于它可以和 colordrop 包放在一起。

这个事件包含一个属性：一个触发该事件的方块对象的引用。同样的，我们也需要创建一个标识符来使得 Block 类和 ColorDrop 类可以找到这个事件。这个标识符应该有 eventClickBlock 的值，并且它将被存储在一个名为 EVENT\_CLICK\_BLOCK 的静态属性中。

```
package com.efg.games.colordrop
{
    import flash.events.*;

    public class CustomEventClickBlock extends Event
```



```
{
    public var block:Block;
    public static const EVENT_CLICK_BLOCK:String = "eventClickBlock";

    public function CustomEventClickBlock(type:String, block:Block,
        bubbles:Boolean=false, cancelable:Boolean=false)
    {
        super(type, bubbles, cancelable);
        this.block = block;
    }

    public override function clone():Event {
        return new CustomEventClickBlock(type, block, bubbles, cancelable)
    }

    public override function toString():String {
        return formatToString(type, "type", "bubbles", "cancelable", "eventPhase");
    }
}
```

## 使用 level 类来控制游戏的难度

在这本书之前的部分我们已经展示了很多不同的方法来创建游戏难度级别，对于魔法色块这个游戏来说，我们将进一步的详细描述这些创意。一个简单的难度等级列表如下所示。在 ColorDrop 类里我们将创建一个这些类的数组来详细描述游戏的难度级别，它是拥有固定斜率的难度等级规则。

每一个难度级别拥有下面的三个属性：

- **allowedColors**：这个数组保存着方块所有可能的颜色值。
- **startPlays**：这个属性表示玩家在当前难度等级玩的次数。当次数超出当前难度等级允许的上限时，游戏结束。
- **scoreThreshold**：只有分数达到当前级别要求的分数时才能进入下一级别。分数必须在 **startPlays** 等于 0 之前达到当前级别要求的分数。

DifficultyLevel 类在构造函数中通过传参方式接收这些属性。

```
package com.efg.games.colordrop
{
```



```
public class DifficultyLevel {

    public var allowedColors:Array;
    public var startPlays:Number;
    public var scoreThreshold:Number;

    public function DifficultyLevel(allowedColors:Array, startPlays:Number,
scoreThreshold:Number)
    {
        this.allowedColors = allowedColors;
        this.startPlays = startPlays;
        this.scoreThreshold = scoreThreshold;
    }
}
}
```

## 创建 GameState 类

GameStates 是这样的一个类：它将包含一些控制游戏流程的静态常量属性。这些值都是游戏状态机的局部属性。我们要为这个游戏创建一个局部的状态机，它将与 Main 类中的状态机保持一致，这些状态（状态机）的详细情况我们将在稍后进行解释。对于现在的我们来说，知道我们将要创建更多的东西是一件好事，可以帮助我们更好更全面地了解整个游戏的结构。

```
package com.efg.games.colordrop
{
    public class GameState {
        public static const STATE_INITIALIZING:int = 10;
        public static const STATE_START_REPLACING:int = 20;
        public static const STATE_WAITING_FOR_INPUT:int = 30;
        public static const STATE_REMOVE_CLICKED_BLOCKS:int = 40;
        public static const STATE_FADE_BLOCKS_WAIT:int = 50;
        public static const STATE_FALL_BLOCKS_WAIT:int = 60;
        public static const STATE_END_GAME:int = 70;
        public static const STATE_END_LEVEL:int = 80;
        public static const STATE_WAIT:int = 90;
    }
}
```



}

你也许会想问为什么这些属性的默认值都是 10 的倍数，而我们之前在 Block 类里创建的静态常量的默认值都是 1 的倍数呢。原因很简单。我们在 Block 类里创建的那些静态属性是用来表示方块继承自 TileSheet 的颜色值的。那些帧都是 1 的倍数，所以我们在代码中设定属性的默认值都是 1 的倍数以适应这种情况。然而，在 GameStates 类中的状态可以使任何值，你不必再去数其他的常量就可以使它适合表示另外两种状态。虽然这样的区分各个属性之前的值是没有必要的，但是这么做其实是一个历史传承下来的习惯。这要追溯到当年 BASIC 语言还是最流行的编程语言时，当时代码行数是有限的。以 10 的倍数来计算代码行数在当时是一个普遍的习惯，同样的原因，我们对状态的计数也沿用了 10 的倍数这种习惯——这允许开发者在新增代码行时不用去计算当前的值应该是多少，只需要知道行数就可以了。现实中，你可以使用任何值来标识这些状态，只要它们都是唯一的。

```
package com.efg.games.colordrop
{
    import flash.display.Sprite;
    import flash.events.*;

    import com.efg.framework.Game;
    import com.efg.framework.CustomEventLevelScreenUpdate;
    import com.efg.framework.CustomEventScoreBoardUpdate;
    import com.efg.framework.CustomEventSound;
    import com.efg.framework.TileSheet;

    public class ColorDrop extends com.efg.framework.Game
    {

        private var gameWidth:int;
        private var gameHeight:int;
```

现在我们将定义一些对游戏非常重要的静态常量。如果你想重用这些代码到其他的项目中，那么有更多的常量值需要你去了解。这些值将定义屏幕中方块的位置、尺寸和网格的空间。

//Game Constants 游戏常量

我们首先要讨论的两个常量是 X\_PAD 和 Y\_PAD。这两个常量的值表示屏幕中的方块在网格中的起始位置 (x, y)。这两个常量将被添加到每一个我们创建的方块中去，这样我们就能将方块准确地定位在网格中了。

```
private static const Y_PAD:int =      50;
private static const X_PAD:int =      135
```

ROW\_SPACING 和 COL\_SPACING 这两个常量表示屏幕中方块之间的行间距和列间距。



```
private static const ROW_SPACING:int =      2;
private static const COL_SPACING:int =      2;
```

BLOCK\_HEIGHT 和 BLOCK\_WIDTH 这两个常量表示每个方块的尺寸。它们也会被用于我们创建 TileSheet 对象时表示每一块图案的尺寸。

```
private static const BLOCK_HEIGHT:int=      32;
private static const BLOCK_WIDTH:int =      32;
```

BLOCK\_ROWS 和 BLOCK\_COLS 表示屏幕中有多少行和多少列的方块将被显示出来。这个游戏是用代码写出来的，你可以通过修改这两个值非常容易地去改变在屏幕中显示多少方块。

```
private static const BLOCK_ROWS:int  =      10;
private static const BLOCK_COLS:int  =      10;
```

下面这些变量的设置将被用于游戏内部的逻辑控制。我们之前已经在“为 Main 添加记分牌”那部分讨论过了它们，不过还是让我们来回顾一下：

- score 变量时玩家得到的游戏总分，将会通过记分牌对象反馈给玩家。
- level 变量时当前游戏的难度级别，也同样会通过记分牌对象反馈给玩家。
- plays 变量是玩家剩余的点击次数，它同样通过记分牌对象反馈给玩家。
- levelScore 变量是玩家当前级别累计的分数，同样通过记分牌反馈给玩家。在 plays 变量的值耗尽之前，这个变量的值必须要大于等于 DifficultyLevel.scoreThreshold 的值，玩家才可以晋级至下一等级。

```
private var score:int;
private var level:int;
private var plays:int;
private var levelScore:int;
```

既然我们打算通过将 DifficultyLevel 对象存储在一个数组中的方法来管理游戏的难度，那么我们就需要定义一些变量来保存这些值。我们创建 difficultyLevelArray 这个变量来保存所有的 DifficultyLevel 对象。currentLevel 变量将保存 DifficultyLevel 类，它表示玩家当前的难度等级。

```
private var difficultyLevelArray:Array;
private var currentLevel:DifficultyLevel;
```

现在我们要声明一些变量来表示玩家在游戏中点击的方块。clickedBlocksArray 这个数组被用来保存玩家点击一个方块后计算得到的它周围相同颜色的方块。board 是一个多维数组，它被用来保存屏幕中方块的逻辑相对位置 (row, col)。这样就可以使我们更简单地根据它周围的方块去计算一个方块了。tempBlock 变量是一个实例属性，它会多次出现在我们的 Game 类里。我们声明这个变量来保存我们每次创建方块时的开销。

```
private var clickedBlocksArray:Array;
private var board:Array;
```



```
private var tempBlock:Block;
```

下面我们设置一些变量被用于游戏的状态机。gameState 变量表示当前游戏的状态。nextGameState 变量和 GameStates.STATE\_WAIT 一起来表示上一个 GameStates.STATE\_WAIT 状态已经结束。

GameStates.STATE\_WAIT 是一个特殊的状态，它让游戏暂停几帧使得画面切换更为平滑。framesToWait 和 framesWaited 这两个变量是和 GameStates.STATE\_WAIT 联系在一起的。

```
private var gameState:int = 0;
private var nextGameState:int = 0;
private var framesToWait:int = 0;
private var framesWaited:int = 0;
```

接下来，我们定义一些难度的设置。在游戏中我们只有两个关于难度的变量，剩下的设置我们都放到了 DifficultyLevel 类里了。然而，这些设置是通用的。首先，POINT\_PER\_BLOCK 就是玩家点击一个方块后匹配的那些方块中每一个方块所得的分值。其次，BONUS\_POINTS\_PER\_BLOCK 是玩家每次单击方块成功获得的分数减去第一次得到分数的值。这个实际上就是超过第一次点击的方块后，每一个方块多加 25 点分值。

```
private static var BONUS_PER_BLOCK:Number = .25;
private static var POINTS_PER_BLOCK:Number = 1;
```

Finally, we define the tileSheet variable we will be using for this game.

最后我们来定义被用于游戏中的 tileSheet 变量。

```
//***** Flex *****
//[Embed(source = 'assets/colordropassets.swf', symbol = 'ColorSheet')]
//private var ColorSheet:Class;
//***** End Flex *****
```

```
private var tileSheet:TileSheet;
```

这个游戏的构造函数和上一个游戏的构造函数很相似。构造函数接收两个参数 gameWidth 和 gameHeight。然后我们给 gameState 赋值为 GameStates.STATE\_INITIALIZING，并且调用 init 方法来给属性赋值。我们设置 gameState 来确保当初始化方法被调用时 runGame 方法可以被调用。切记，runGame 方法它会被 Main 类的计时器监听方法所调用，所以我们不必从游戏中控制它；游戏自己就会知道该做什么。

```
public function ColorDrop(gW:int,gH:int) {
    gameWidth=gW;
    gameHeight=gH
    init();
    gameState = GameStates.STATE_INITIALIZING;
}
```



## 初始化游戏

init 方法被游戏的构造函数调用来设置一些变量的初始值。我们从设置 tileSheet 这个变量的值开始说起。就如我们在之前段落讨论过的 Block 类一眼，当 方块被创建的时候我们需要将 tileSheet 传给 Block 实例。TileSheet 的是一个我们在 SWF 库（在这里实例中，它指代的就是 ColorSheet）里设置好的位图对象，并且它的高度和宽度我们将通过在 tile sheet 中切割来获取。Flex 版本的代码已经被注释掉了，但只有这样你才能看到不同之处。

```
public function init():void {  
    //***** Flash IDE *****  
    tileSheet = new TileSheet(new ColorSheet(0,0), BLOCK_WIDTH,BLOCK_HEIGHT);  
    //***** End Flash IDE *****  
    //***** Flex *****  
    //tileSheet = new TileSheet(new ColorSheet().bitmapData, BLOCK_WIDTH,  
    BLOCK_HEIGHT);    //***** End Flex *****  
}
```

现在我们需要为游戏创建不同的级别。为了做到这一点，我们首先要初始化我们的 difficultyLevelArray 数组。然后，我们就可以通过向数组添加 DifficultyLevel 实例非常容易地创建一个级别。

由于游戏的目标是消除相邻的同颜色的方块，游戏难度的斜率就通过使方块拥有更多颜色来实现。我们通过传递一个 Block.BLOCK\_COLOR\_XXX 数组中的值作为 DifficultyLevel 的首个参数来实现这个。另外一个参数就是 startPlays，当某个难度级别开始的时候游戏的可玩儿次数 就会被添加进来；而 scoreThreshold，分数必须要达到该级别的要求分数才能晋级到下一级别。这么做就使得每个级别这些变量的值都是相同的。这 是我们游戏首先需要修正的地方，试着改变一下取值标准来看下这个游戏的难度是否可以用其他的方式来管理。

```
difficultyLevelArray = new Array();  
difficultyLevelArray.push(new DifficultyLevel(  
    [Block.BLOCK_COLOR_RED,Block.BLOCK_COLOR_GREEN,Block.BLOCK_COLOR_BLUE],  
    10,500));  
difficultyLevelArray.push(new DifficultyLevel(  
    [Block.BLOCK_COLOR_RED,Block.BLOCK_COLOR_GREEN,Block.BLOCK_COLOR_BLUE,  
    Block.BLOCK_COLOR_VIOLET],10,500));  
difficultyLevelArray.push(new DifficultyLevel(  
    [Block.BLOCK_COLOR_RED,Block.BLOCK_COLOR_GREEN,Block.BLOCK_COLOR_BLUE,  
    Block.BLOCK_COLOR_VIOLET,Block.BLOCK_COLOR_ORANGE],10,500));  
difficultyLevelArray.push(new DifficultyLevel(  
    [Block.BLOCK_COLOR_RED,Block.BLOCK_COLOR_GREEN,Block.BLOCK_COLOR_BLUE,  
    Block.BLOCK_COLOR_VIOLET,Block.BLOCK_COLOR_ORANGE,Block.BLOCK_COLOR_Y  
    ELLOW],
```



```
10,500));  
difficultyLevelArray.push(new DifficultyLevel(  
    [Block.BLOCK_COLOR_RED,Block.BLOCK_COLOR_GREEN,Block.BLOCK_COLOR_BLUE,  
    Block.BLOCK_COLOR_VIOLET,Block.BLOCK_COLOR_ORANGE,Block.BLOCK_COLOR_Y  
    ELLOW,  
    Block.BLOCK_COLOR_PURPLE],10,500));  
  
}
```

newGame 方法被 Main 类调用。对这个游戏而言，我们设置 level，score 和 plays 变量的值为 0。这就是我们要为新的游戏所做的一切。

```
override public function newGame():void {  
    level = 0;  
    score = 0;  
    plays = 0;  
}
```

和 newGame 方法不同的是，newLevel 方法有一些有趣的代码。我们通过更新 level 和 plays 变量的值来开始调用 newLevel 方法，然后 游戏开始。对于级别这个变量，我们使它的值增加，并且检测它的值是否比我们存在 difficultyLevelArray ( difficultyLevelArray.length - 1 ) 的最终等级的值大。如果是这样的话，我们从 difficultyLevels 数组中获取最后一个有效级别的值。我们使用一个叫做 tempLevel 的变量来保存局部的等级，因为当我们想为 ScoreBoard 对象保留当前的级别，我们必须读取最后一个有效的级别。

下一步，我们从 tempLevel 到 plays 变量添加 startPlays 的值。切记，玩家可以得到为当前级别到最终级别保留的可玩次数。然后我们设置 levelScore 的值为 0。levelScore 仅仅用来表示玩家当前级别所累积的得分。我们使用这个变量来判断玩家是否已经得到或者超过当前级别的 scoreThreshold，并且晋级下一级别，我们派发事件来通报 plays，level 和 threshold 的新值。

```
override public function newLevel():void {  
    level++;  
    var tempLevel:int = level;  
    if (tempLevel > (difficultyLevelArray.length-1)) {  
        tempLevel = difficultyLevelArray.length-1  
    }  
    currentLevel = difficultyLevelArray[tempLevel-1];  
    plays += currentLevel.startPlays;  
    levelScore = 0;  
    dispatchEvent(new CustomEventLevelScreenUpdate(  
        CustomEventLevelScreenUpdate.UPDATE_TEXT, String(level)));
```



```
dispatchEvent(new CustomEventScoreBoardUpdate(
    CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_PLAYS,String(pl
ays)));
dispatchEvent(new CustomEventScoreBoardUpdate(
    CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_LEVEL,String(le
vel)));
dispatchEvent(new CustomEventScoreBoardUpdate(
    CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_THRESHOLD,
    String(currentLevel.scoreThreshold)));
```

现在，我们需要初始化屏幕上的方块。我们要将所有屏幕上方块的引用存储在一个叫做 `board` 的二维数组中。首先，我们初始化这个数组。接下来我们创建两个个 嵌套的 `for` 循环。第一层 `for` 循环为每一排方块建立一个新的数组。第二层 `for` 循环在每个数组的每一个行列组合的节点上建立一个 `null` 值 (`board[r][c] = null;`)。我们将会利用这些 `null` 值来决定我们是否需要创建一个方块并把它放到属于它的位置上。然后我们初始化 `clickedBlockArray`。就像我们之前 谈到过的那样，我们会用这个数组来保存玩家点击方块的同色方块列表。最后，我们将 `gameState` 的值设定为 `GameStates.STATE_START_REPLACING`，这实际上是将方块放置于屏幕上的操作。

```
board = new Array();
for (var r:int = 0; r < BLOCK_ROWS; r++) {
    board[r] = new Array();
    for (var c:int = 0; c < BLOCK_COLS; c++) {
        board[r][c] = null;
    }
}
clickedBlocksArray = new Array();
gameState = GameStates.STATE_START_REPLACING;
}
```

## 为 ColorDrop 添加状态机。

在 我们谈及将方块置于屏幕之前，我们应该先讨论一下我们将要在 `ColorDrop` 类中创建的状态机。我们已经为 `Main` 类创建了一个状态机。它控制着游戏，并告知何时开始一个新游戏，开始一个新的级别，并且通过调用 `runGame` 方法来运行游戏。到目前为止，这对我们的游戏来说已经足够了。然而，当我们谈到 休闲益智型游戏，引入状态机完善 `runGame` 方法有利于下一步的工作流程。在你合上书并尖叫着跑开之前，让我们先把它解释清楚。

如果你能回忆起第一章，我们用 `switch` 语句创建了一个非常简单的状态机。我们揭示了为什么我们认为这是一个好主意且用代码实际演示了一回。回过头我们再 看一下，我们相信状态机帮助我们保持了游戏的逻辑状态流并且需要很多布尔值的变量和混乱的检测判断来确保它们都被赋予正确的值。在第二章里我



们进一步晚上 了那个状态机，使得它通过摆脱 switch 语句和使用用来调用当前状态的 systemFunction 变量来运行的更有效率。

我们的魔法方块游戏类需要同样的状态机。原因很简单：我们需要在游戏流程上拥有更多的控制。既然，游戏的大多数时间都是处于等待用户输入的状态中，我们打算 限制那些在任何给定时间都可以发生的流程。举例来说，当方块落到屏幕当中时，我们要确保用户可以点击它们。如果游戏中没有状态机，我们就使 Main 类有了 同样的麻烦：太多的布尔值来检测和保持值的更新。用一个简单的状态机，所有我们需要做的仅仅是检测当前的 gameState 来决定什么样的输入是有效 的，什么是游戏接下来该做的。

对于魔法方块这个游戏来说，我们打算建立一个与第一章中使用 switch 语句建立的状态机一样的简单的时间机。我们决定这样做是为了告诉你，当前最先进的游 戏制作的技巧，那就是使用 switch 分支语句风格的状态机可以变得和 systemFunction 风格的状态机一样有用。

回忆一下，在我们的 GameState 类里定义了一些状态属性。

```
public static const STATE_INITIALIZING:int          = 10;
public static const STATE_START_REPLACING:int      = 20;
public static const STATE_WAITING_FOR_INPUT:int     = 30;
public static const STATE_REMOVE_CLICKED_BLOCKS:int = 40;
public static const STATE_FADE_BLOCKS_WAIT:int     = 50;
public static const STATE_FALL_BLOCKS_WAIT:int     = 60;
public static const STATE_END_GAME:int             = 70;
public static const STATE_END_LEVEL:int            = 80;
public static const STATE_WAIT:int                 = 90;
```

我们就是用这些状态来控制游戏流的，这种方法类似于在一种编程语言 ( AS3 ) 中创造了一个小型的抽象编程语言。你可能会注意到一点就是我们尝试 ( 尽可能地 ) 利用 switch 分支语句来控制状态的改变。用这种方式，跟踪游戏流是很容易的。然而我们不能总是在方法中改变状态，但这是我们的目标。

GameStates.STATE\_INITIALIZING 被放置在 ColorDrop 类的构造函数里，并且当游戏开始它对所有发生的事情是空置的状态，与此同时，Main 类正在调用 newGame 和 newLevel 方法。

```
override public function runGame():void {
    switch(gameState) {
        case GameStates.STATE_INITIALIZING:
            break;
```

GameStates.STATE\_START\_REPLACING 被放置在当方块需要被放置在屏幕——每一个级别开始时和玩家点击方块并从网格中消除方块时的条件限定中。我们使这个状态标准化使得它可以运行在所有的方块实例中。replaceBlocks 方法仅仅是用来获取屏幕上的方块对象，但是这些方块仍然需要被落到 制定的位置。这就是 GameStates.STATE\_FALL\_BLOCKS\_WAIT 常量存在的意义。我们将 GameStates.STATE\_WAITING\_FOR\_INPUT 常量赋值给 nextGameState 变量。当



GameStates.STATE\_FALL\_BLOCKS\_WAIT 通过计算了解所有的方块已经在屏幕中停止落下动作时常量 GameStates.STATE\_FALL\_BLOCKS\_WAIT 将赋值给 gameState。

```
case GameStates.STATE_START_REPLACING:
    replaceBlocks();
    nextGameState = GameStates.STATE_WAITING_FOR_INPUT;
    gameState = GameStates.STATE_FALL_BLOCKS_WAIT;
    break;
```

当玩家正在考虑选择哪个方块时，GameStates.STATE\_WAITING\_INPUT 就是一个所谓的普通状态。如果这是一个计时游戏，你或许会在 这里更新计时器。我们调用一个 checkForEndLevel 的方法，因为它很适合被放到这里。在玩家点击方块之后，这个调用就会被正确的执行，但是我们把它放到这里是因为当大多数控制游戏流的方法都在游戏流中时这更容易被理解它的含义。

```
case GameStates.STATE_WAITING_FOR_INPUT:
    checkforEndLevel();
    break;
```

GameStates.STATE\_REMOVE\_CLICKED\_BLOCKS 是一个有着非常具有描述性名字的状态。这个状态被放置到在当方块被从屏幕中移除时。这个状态与 GameStates.STATE\_WAIT(将在稍后对它 做出描述)有一定的关系。我们创建这个状态使得在当玩家点击一个方块和方块从屏幕中移除之间有一个短小的暂停。这给每一次点击带来了更生动的效果，正如玩 家看到的那样，所有被连接的方块在它们被移除之前高亮显示。我们调用 removeClickedBlock (在“为屏幕添加方块”段落中对此方法有所描述)来摆脱当前点击的那个方块，并且将 GameStates.STATE\_START\_REPLACING 的值赋给 gameState 来使更多的方块回到屏幕中来。

```
case GameStates.STATE_REMOVE_CLICKED_BLOCKS:
    removeClickedBlocks();
    gameState = GameStates.STATE_START_REPLACING;
    break;
```

GameStates.STATE\_FADE\_BLOCKS\_WAIT 被放置到当整个一组方块通过设置它们的 alpha 属性来从屏幕中淡出消失时。这个状态保持始终调用 checkForFadeBlocks 方法，这样当所有 方块都从屏幕中淡出消失之后，他才能够将 nextGameState 的值赋给 gameState。

```
case GameStates.STATE_FADE_BLOCKS_WAIT:
    if (!checkForFadingBlocks()) {
        gameState = nextGameState;
    }
    break;
```



GameStates.STATE\_FALL\_BLOCKS\_WAIT 被放置到当方块正在落在屏幕或者从屏幕离开时，这通常发生在级别刚开始，方块被点击和游戏结束时。这个方法会继续调用 checkForFallingblocks 方法，所以当所有的方块停止落下时它可以将 nextState 赋值给 gameState。

```
case GameStates.STATE_FALL_BLOCKS_WAIT:
    if (!checkForFallingBlocks()) {
        gameState = nextState;
    }
    break;
```

GameStates.STATE\_END\_LEVEL 被放置到当 ColorDrop 断定玩家 levelScore 的值大于等于 currentLevel.scoreThreshold 时。endLevel 方法被调用且我们将 GameStates.STATE\_INITIALIZING 赋值给 gameState，这样我们就可以等待 Main 类调用 nextLevel 方法了。

```
case GameStates.STATE_END_LEVEL:
    endLevel();
    gameState = GameStates.STATE_INITIALIZING;
    break;
```

GameStates.STATE\_END\_GAME 被放置到当 ColorDrop 断定 plays 属性等于 0，但是 levelScore 不等于或者大于 currentLevel.scoreThreshold。我们调用 endGame 方法，它将会使游戏停止。

```
case GameStates.STATE_END_GAME:
    endGame();
    break;
```

我们创建的 GameStates.STATE\_WAIT 是一个特殊的状态，它允许我们在继续游戏之前的指定的帧数里暂停游戏。这听起来似乎没什么作用，但是对于实现更生动的效果，它是非常有用的。我们来论述一下我们一会儿用它来做什么，但是对于现在而言，我们只是谈论一下它的工作原理。在 gameState 被赋值为 GameStates.STATE\_WAIT 之前，framesWaited 变量会被置为 0。相反地，waitFrames 被置为最大帧值来暂停游戏。并且，nextGameState 将被赋值为 STATE\_XXX 的其中一个值，这样在 GameStates.STATE\_WAIT 状态完成之后我们状态就发生了改变。

```
case GameStates.STATE_WAIT:
    framesWaited++;
    if (framesWaited >= framesToWait) {
        gameState = nextState;
    }
    break;
}
```



最终，在我们 switch 分支语句的外面，我们调用 render 方法和 update 方法。如果我们不这样做，我们就需要在那些条件语句中被调用的方法中更多次的调用 update 方法和 render 方法。为了更有效，我们做一些改变来进行更新数据和渲染场景操作，这样做可以保证它们只在需要服务的对象中被调用。

```
update();
render();

}
```

当我们需要查找发现当前屏幕中是否有方块正在移动时，checkForFallingBlocks 方法将被调用。这个方法在 board 这个二维数组中循环遍历所有的方块，如果哪个方块正在移动（isFalling = true），它就会返回 false。

```
public function checkForFallingBlocks():Boolean {
    var isFalling:Boolean = false;
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempBlock = board[r][c];
            if (tempBlock.isFalling) {
                isFalling = true;
            }
        }
    }
    return isFalling;
}
```

当我们需要查找发现当前屏幕中是否有方块正在淡出中时，checkForFadingBlocks 方法将被调用。这个方法也会在 board 这个二维数组中循环遍历，如果哪个方块正在淡出（isFading = true），它就会返回 false。

```
public function checkForFadingBlocks():Boolean {
    var isFading:Boolean = false;
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempBlock = board[r][c];
            if (tempBlock.isFading) {
                isFading = true;
            }
        }
    }
}
```



```
return isFading;
```

```
}
```

## 在屏幕中添加方块

当 gameState 的值等于 GameStates.STATE\_START\_REPLACING 的值时，runGame 方法调用 replaceBlocks 方法将方块对象放置在屏幕上。我们要写一个对我们所有游戏都通用的方法：无论方块的位置是否需要被取代（在一个级别刚刚开始的时候或者在用户点击并移除方块 自后），它都使用相同的方法操作。我们替换方块的操作方法非常简单：我们循环遍历存储在 board 的二维数组，然后当我们找到一个行列组合的值为 null ( board[r][c] == null )，我们就调用 addBlock(r, c)方法来填充它。就是这样。addBlock 方法做了一项多么有趣的工作啊。

```
public function replaceBlocks():void {  
    for (var r:int = 0; r < board.length; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            if (board[r][c] == null) {  
                board[r][c] = addBlock(r,c);  
            }  
        }  
    }  
}
```

一打眼看过去 addBlock 方法看起来似乎挺复杂的，但是实际上它很简单。代码看起来很难是因为所有需要被计算的定位方块的常量。让我们先把简单的部分摆脱掉：找到方块的颜色。我们从 DifficultyLevel 类定义的数组 CurrentLevel.allowedColors 中得到一个随机的颜色。首先，我们发现 allowedColors 数组的索引像下面这样：

```
var randomColor= Math.floor(Math.random()*currentLevel.allowedColors.length);
```

然后，我们这个索引来存储颜色，以便后面使用：

```
var blockColor = currentLevel.allowedColors[randomColor];
```

现在我们需要创建一个方块的实例对象，并且将它放到 tempBlock 对象中。下面是参数的列表，我们将它们传给 Block 类的构造函数。

- blockColor：我们从 currentLevel.allowedColors 数组中得到的颜色
- tileSheet：持有方块图形的 tile sheet 的引用
- row：从 replaceBlocks 得到的在 board 数组中的行索引
- col：从 replaceBlocks 得到的在 board 数组中的列索引
- (row\*BLOCK\_HEIGHT)+Y\_PAD+(row\*ROW\_SPACING)：赋值给 Block 对象的 fallEndY 属性



$row * BLOCK\_HEIGHT$  的值对于方块对象来说就是行位置的 y 轴坐标值。Y\_PAD 表示的是方块在网格中从屏幕顶端开始算起到方块位置的垂直距离。 $row * ROW\_SPACING$  的值表示方块的行间距。把这些值加到一起就得到了方块在屏幕中的位置数据。

```
public function addBlock(row:Number, col:Number):Block {  
    var randomColor:Number =  
        Math.floor(Math.random()*currentLevel.allowedColors.length);  
    var blockColor:Number = currentLevel.allowedColors[randomColor];  
    tempBlock = new Block(blockColor, tileSheet, row, col,  
        (row*BLOCK_HEIGHT)+Y_PAD+(row*ROW_SPACING),(Math.random()*10)+10 );
```

接下来，我们设置方块对象的 x 属性的初始值，它是和 fallEndY 值最接近的，除非我们用 BLOCK\_WIDTH, X\_PAD, and COL\_SPACING 这几个值也加入到计算当中。y 属性的初始值很明显的就是  $0 - BLOCK\_HEIGHT$ ，所以方块才能从屏幕的外围开始落下。图 8-3 给我们展示了在魔法色块这个游戏中这些常量怎么计算方块盖在何处开始和结束的。

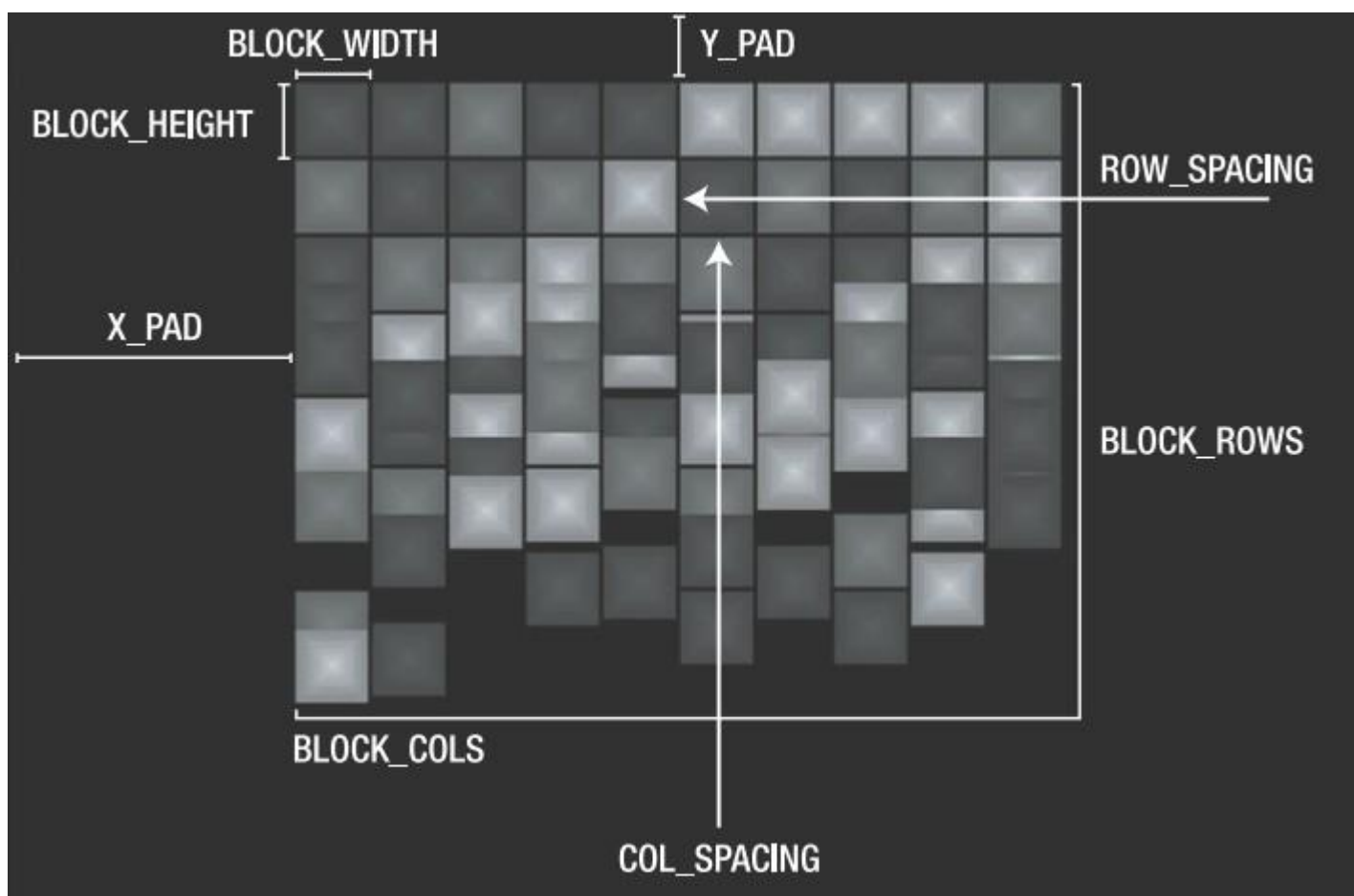


图 8-3. 方块落到指定位置（附常量注解）

接下来，我们为之前 CustomEventClickBlock 类里定义的 CustomEventClickBlock.EVENT\_CLICK\_BLOCK 事件添加一个监听方法。我们将把 CustomEvent 类命名



为 CUSTOMEVENT\_CLICKBLOCK。当用户点击时，这个事件也同样需要被派发出去。blockClickListener 方法将被用来处理这个事件。最终，我们将通过 addChild 方法将方块添加到显示列表中且返回一个方块对象的引用给调用者。在我们的范例中，它总是属于存储方块引用的 board 二维数组的。

```
tempBlock.x=(col*BLOCK_WIDTH)+X_PAD+(col*COL_SPACING);
tempBlock.y= 0 - BLOCK_HEIGHT;
tempBlock.addEventListener(CustomEventClickBlock.EVENT_CLICK_BLOCK,
    blockClickListener, false, 0, true);
this.addChild(tempBlock);
return tempBlock;
}
```

## 扩展游戏的 update 和 render 方法

这个游戏的 update 方法和 render 方法和书中另外一个游戏很相似。尽管如此，我们仍然需要扩展一下它们的域逻辑来协助 ColorDrop 的状态机。回想一下前面的章节，这两个方法的调用是和 gameSate 变量之间毫无关系的。我们这么做事因为很多的状态需要这两个方法被调用，但是在 state 的条件分支语句中调用它们就太混乱了。取而代之的是，我们加入一些简单的域逻辑来检测方块对象是否需要它们调用 update 方法和 render 方法。

为了刷新，我们循环遍历在 board 二维数组中的所有方块对象找到值不是 null 的。这个细节很重要，因为方块仍然能够在网格还没有全部被新的方块对象填满的时候落入（落入）指定的位置。如果我们不在数组中检测 null 值，游戏将会因为空对象的引用在运行时抛出异常。如果方块不为 null，那么我们检测 tempblock.isFalling 的值是否为 true。如果是，我们就调用 tempBlock.update 方法。不是的话就什么也不用做了。

```
public function update():void {
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempBlock = board[r][c];
            if (tempBlock != null) {
                if (tempBlock.isFalling) {
                    tempBlock.update();
                }
            }
        }
    }
}
```



render 方法和 update 方法非常相似。首先我们检测 null 值，然后，如果 tempBlock.isFalling 活或者 tempBlock.isFading 的值等于 true 我们就调用 tempBlock.render 方法。我们不在 update 方法中判断 tempBlock.isFading 的值，因为 tempBlock.update 方法不是用来处理方块的淡出特效的。

```
public function render():void {
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempBlock = board[r][c];
            if (tempBlock != null) {
                if (tempBlock.isFalling || tempBlock.isFading) {
                    tempBlock.render();
                }
            }
        }
    }
}
```

## 等待用户输入

GameStates.STATE\_WAITING\_FOR\_INPUT 这个游戏状态是当 ColorDrop 在游戏循环中等待鼠标点击时设置的。当一个方块被鼠标点击之后，CustomEventClickBlock.EVENT\_CLICK\_BLOCK 事件将被派发，并且 blockClickListener 方法将被调用来处理这个事件（详见图 8-4）。

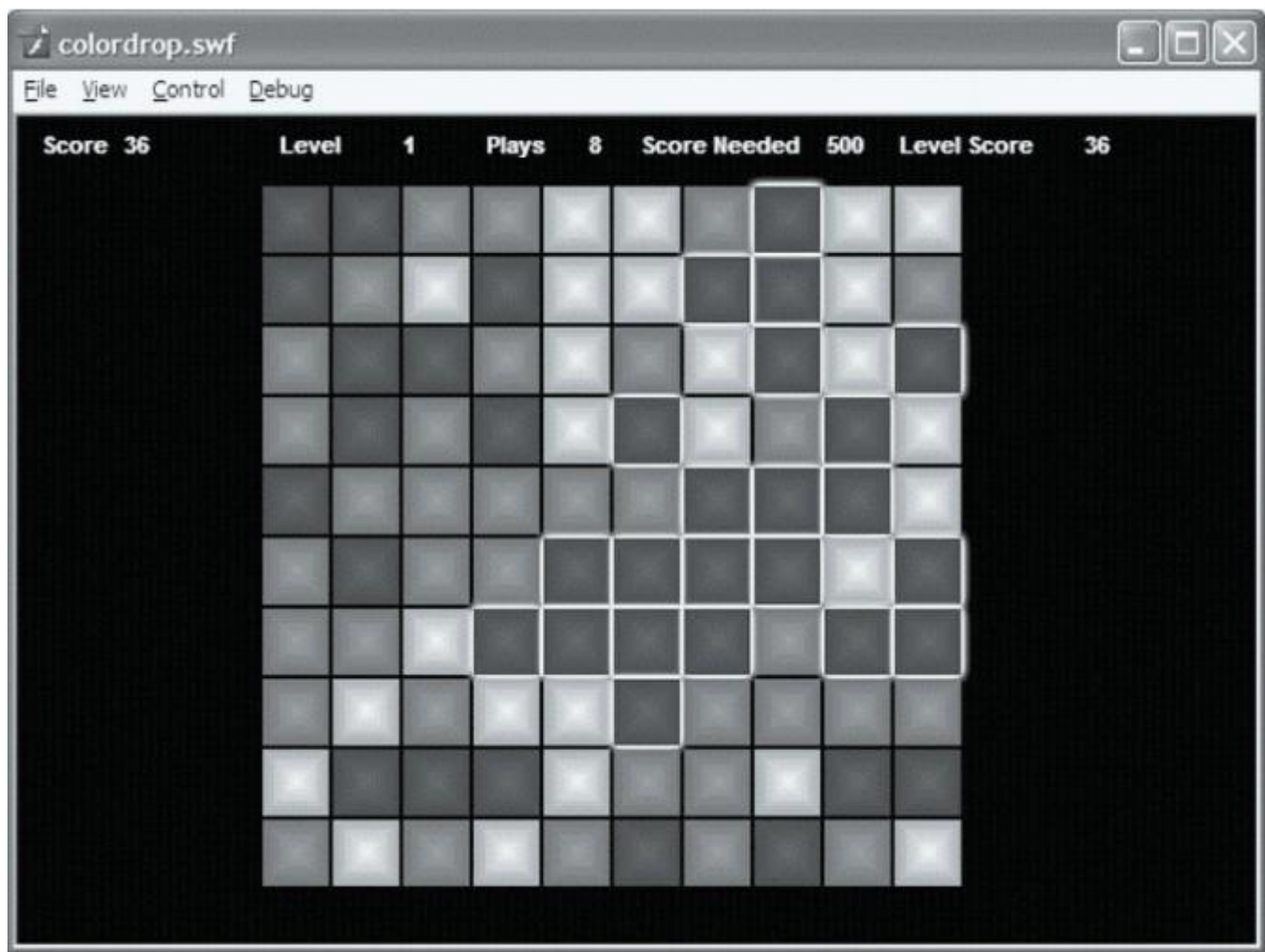


图 8-4. 点击一个带颜色的方块，并且选中了所有与它相连的同颜色方块。

在 `blockClickListener` 方法中我们首先要做的是判断游戏状态是否处于 `GameStates.STATE_WAITING_FOR_INPUT`。如果是，我们加入代码的主要功能就是要派发 `CustomEventSound.PLAY_SOUND` 事件来播放 `Main.SOUND_CLICK` 音效。这会给用户鼠标点击操作一个声音上的反馈。

然后我们设置所有必要的变量来创建 `GameStates.STATE_WAIT` 状态 (`framesToWait = 15`, `framesWaited = 0` 和 `nextGameState = GameStates.STATE_REMOVE_CLICKED_BLOCKS`)，`plays` 变量的值将减少(因为玩家点击了一个方块)，且设置 `gameState` 变量的值为 `GameStates.STATE_WAIT`。这样将会在用户点击方块，匹配的方块被高亮显示，和游戏状态为 `GameStates.STATE_REMOVE_CLICKED_BLOCKS` 时所有匹配方块从屏幕上被移除这几个操作之间产生一个生动的暂停效果。

```
public function blockClickListener(e:CustomEventClickBlock):void {  
    if (gameState == GameStates.STATE_WAITING_FOR_INPUT) {  
        dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND,  
            Main.SOUND_CLICK,false,1,0,1));  
    }  
}
```



```
tempBlock = e.block;
clickedBlocksArray = findLikeColoredBlocks(tempBlock);
framesToWait = 15;
framesWaited = 0;
plays--;
nextGameState = GameStates.STATE_REMOVE_CLICKED_BLOCKS;
gameState = GameStates.STATE_WAIT;
}
}
```

## 用非递归方法来检测方块

这么说吧，这一节我会教您一些关于递归方法的知识，并且告诉您在游戏中怎样使用递归会更有意义。递归方法就是可以反复地调用自身来重复某个操作的方法。你可以想象，像魔法色块这样的游戏，递归方法是非常有用的。我们需要检测很多方块来判断它们是否连接在一起。为了做到这一点，我们需要调用一些方法，并且它们需要非常聪明地决断出一个方块是否和原始方块有连接。

当你在游戏中点击一个方块时，ColorDrop 类需要查找所有在被点击方块对象周围的同色方块对象。

举例来说，我们可以创建一个名为 testBlock 的方法来传递一个 Block 对象，并且它的职责是为每一个连接到的方块调用 testBlock 方法。如果我们这么做，看看执行栈，你会发现下面这样的情况：

```
testBlock()
  testBlockColor()
    testBlock()
      testBlockColor()
        testBlock()
          testBlockColor()
            testBlock()
```

这个递归会持续进行下去，直到所有同色相连的方块都被检测过。除了比较占用资源外，这也不是什么大不了的事儿。在大多数设备上，调用方法是通过向 CPU 内部的栈传递参数来完成的。当设备上只有少量内存（例如，iPhone 手机或者其他移动设备），调用栈变得越庞大，可用内存的就会越快被耗光。与此同时，非官方的测试结果显示（在其他平台上）递归操作要比其他的操作更慢，比如迭代法（举个例子，详见：<http://shiman.wordpress.com/2008/05/28/recursion-vs-iteration/>）。不管怎样，即使递归操作比迭代操作更快，我们游戏也有一个不使用它的更好的原因。在 AS3 中，递归调用被限定在 256 层以内。也就是说，大多数情况下，屏幕上方块最多只有 256 个方块可以被测试到（意味着只要是同颜色的方块就会被递归调用一次）。即使魔法色块这个游戏一次只用 100 个方块，理论上一个类似的游戏都应该超过 256 个。使用 AS3 的递归调用就必须限制游戏中方块的数量这实在说不过去。



取而代之的是，我们打算创建一个简易的方法来反复的检测每一个方块对象。我们通过使用三个平行的数组对象来完成这个任务。这些数组将帮助我们限制游戏中我们需要检测方块的数量并且帮助我们跟踪检测中匹配的和不匹配的方块。下面就是我们打算要创建的数组：

- `blocksToCheck` :这是一个我们判定与点击方块同颜色且毗邻的或者与毗邻方块同颜色且毗邻的方块对象的列表。
- `blocksMatched` : 这个数组包含了经过我们判断符合条件的方块对象。
- `blocksTested` : 这个数组包含了我们所有已经过检测的方块对象。

为什么我们需要那三个数组呢？它们只是用来保证我们的迭代方法的检测次数是必须的，且不能一直检测哪些方块匹配，哪些已经被检测过了这样进行下去（浪费可用资源）。

```
public function findLikeColoredBlocks(blockToMatch:Block):Array {  
    var blocksToCheck:Array = new Array();  
    var blocksMatched:Array = new Array();  
    var blocksTested:Array = new Array();
```

另外两个数组也非常重要。它们表示我们检测的方块在二维数组 `board` 中的行列定位。既然我们允许任何与点击方块相毗邻的方块被检测（或者毗邻方块的毗邻方块，等等），那么我们就需要检测任何一个方块周围的 9 个格子的位置来判断是否与点击方块有相同颜色的方块。图 8-5 图解了我们需要检测的行列位置，以及它们与行列列表值的关系。

```
var rowList:Array = [-1, 0, 1,-1,1,-1,0,1];  
var colList:Array = [-1,-1,-1, 0,0, 1,1,1];
```



图 8-5. 方块周围需要检测的行和列定位

在我们第一次操作中，我们把点击方块对象的 `blockColor` 属性储存在了 `colorToMatch` 中，我们一会儿就会用到它了。

```
var colorToMatch:Number = blockToMatch.blockColor;
```



接下来，我们将 blockToMath（方块作为参数传递给该方法）插入到 blocksToCheck 数组中。我们需要这个数组的头来开始我们的迭代操作。剩下的计算操作会完全由 while 循环语句来接管。

```
blocksToCheck.push(blockToMatch);
```

现在，我们开始大的循环操作。这是一个非常复杂的循环操作，那为什么不让我们先稍微停下来休息一下？深呼吸，好的，也许你应该先喝一瓶饮料然后再来看下面的部分。好了吗？好的，我们开始喽。我们这个大的 while 循环操作将会一直到 blocksToCheck 数组的末端。既然我们用 blockToMatch（用户点击的方块）开了个头，那么我们就从进入循环操作开始吧。在循环中我们要做的第一件事就是从 blocksToCheck 数组中删除最后一个方块，并将它放到 tempBlock 中。然后我们检测 tempBlock 的颜色值（tempBlock.blockColor）来判断它是否和用户点击方块的颜色值相同（colorToMatch）。

既然我们从第一个方块中得到的 colorToMatch 值恰好等于 tempBlock 的值，那么我们的判断结果返回的将会是 true。然后我们将 tempBlock 对象插入到 blocksMatched 数组中（因为它是符合匹配的方块），并且调用 tempBlock.makeBlockClicked 方法，这个方法会在方块对象上添加一个发光滤镜使方块处于高亮显示状态。这个检测是为了保证循环的条件得到的结果是 true，以此可以继续循环下去。然后后续的方块对象也将迭代的进行检测，除了用户点击的那个方块之外。

```
while(blocksToCheck.length > 0) {  
    tempBlock = blocksToCheck.pop();  
    if (tempBlock.blockColor == colorToMatch) {  
        blocksMatched.push(tempBlock);  
        tempBlock.makeBlockClicked();  
    }  
}
```

现在我们打算检测所有在 tempBlock 周围的临近的方块。回想一下，我们之前创建了 2 个数组，rowList 和 colList，并且它们的值域图 8-5 中的方块有关系。首先，我们创建一个临时的 Block 对象，tempBlock2，来进行我们的操作。接下来，我们创建一个 for 循环来重复遍历数组 rowList。既然 rowList 和 colList 都有相同的值，那么我们就没必要遍历数组的时候也检查它了，我们会使用 rowList 数组的索引（i）来迭代 colList 数组。

下一个条件检测 rowList[i]，colList[i] 表示的方块是否在 board 二维数组表示的范围之内。这个值必须要大于 0 小于 BLOCK\_ROWS 和 BLOCK\_COLS（分别为行和列）。如果我们不这样做，运行时就会报“数组访问超出边界”异常错误。

```
var tempBlock2:Block;  
for (var i:int = 0;i < rowList.length;i++) {  
    if ((tempBlock.row + rowList[i]) >= 0 && (tempBlock.row + rowList[i])  
        < BLOCK_ROWS && (tempBlock.col + colList[i]) >= 0 &&  
        (tempBlock.col + colList[i]) < BLOCK_COLS) {
```



所以现在我们知道方块处于 board 二维数组中的合法的范围之内，我们设置 tr 等于 row 的计算值，设置 tc 等于 column 的计算值。现在我们有了两个值 来表示图 8-5 中的一个方块。我们让 tempBlock2 的值等于通过 tr 和 tc 这两个索引所表示的方块对象 ( board[tr][tc] ) 的值。

```
var tr:int = tempBlock.row + rowList[i];
var tc:int = tempBlock.col + colList[i];
tempBlock2 = board[tr][tc];
```

现在 我们已经为更重要的检测做好了准备。如果 tempBlock2 的 blockColor 属性等于 colorToMatch 的值 ( 被点击方块的颜色值 )，那么我们的业务逻辑就完成了。

然而，我们现在还没准备打算把这个方块插入到 blocksToCheck 数组中。我们还需要检测一下这个方块是在 blocksToCheck 里还是在 blocksTested 里。如果我们不做这个检测，同一个方块将会被多次插入到 blocksToCheck 里，并且这个方法永远都会继续运行下去。

一个非常简单的方法来检测一个值是否已经存在于数组中就是使用 Array 类的 indexOf 方法。当我们传递我们把检测的方块对象 ( tempBlock2 ) 作为参数传递给这个方法时，如果它不在数组中它就会返回 -1。这是我们期望看到的返回值。通过在 blocksToCheck.indexOf(tempBlock2) 和 blocksTested.indexOf(tempBlock2) 中检测 -1 这个值，我们就会知道是否该把方块对象插入到 blocksToCheck 数组中。

```
if (tempBlock2.blockColor == colorToMatch &&
    blocksToCheck.indexOf(tempBlock2) == -1
    && blocksTested.indexOf(tempBlock2) == -1) {
    blocksToCheck.push(tempBlock2);
}
}
```

在我们圆满地检测了这个方块之后，我们将它插入到 blocksTested 数组中，然后循环回继续检测 blocksToCheck 数组，直到不再存在方块对象。当循环完成之后，我们将 blocksMatched 数组完整的返回给调用者。

你现在也许会想 “等等，为什么我既需要 blocksToCheck 数组还需要 blocksTested 数组？为什么我不能只用一个数组 blocksTested 就搞定呢？我们之前不是已经看了方块对象的列表了吗？” 好吧，答案是必须的。但是请记住，我们一个循环只需要检测八个方块而已。任何这些方块对象都将被插入到 blocksToCheck 数组中，但是只有一个，tempBlock 对象将被插入到 blocksTested 数组中。不检测 blocksToCheck 数组的话，每次这个方法被执行我们就会进入一个死循环里。因为方块对象的多数实例已经在 blocksToCheck 数组中了，但是不在 blocksTested 数组里，他们需要被插入到 blocksToCheck 数组中。我们将永远无法到达 blocksToCheck 数组的末端。

```
blocksTested.push(tempBlock);
}
return blocksMatched;
```



}

## 移除方块

游戏状态 `GameStates.STATE_REMOVE_CLICKED_BLOCKS` 被设定在当魔法色块这个游戏已经完成了调用 `findLikeColoredBlocks` 方法并且 `GameStates.STATE_WAIT` 已经完成了生动的暂停操作时。现在有一个方块对象的数组叫做 `clickedBlocks`，这个数组可以让我们计算玩家得到的分数，从屏幕中移除这些方块，并且可以用方块对象重新填满 `board` 这个二维数组。从状态机 `switch` 条件分支语句中调用的 `removeClickedBlocks` 方法在 `runGame` 方法中开始这一操作过程。

`removeClickedBlocks` 方法是一个很简单的函数，因为大部分真正的工作都交给了其他的方法来执行。首先，它调用 `removeClickedBlocksFromScreen` 方法来移除屏幕中的方块，然后调用 `moveBlocksDown` 方法来让 `board` 二维数组中剩余的方块移动到新位置。然后我们重新初始化 `clickedBlocks` 数组，为了安全起见，我们已经在 `removeClickedBlocksFromScreen`（在下面的代码片段中展示）方法中移除了这些内容。这之后，我们派发一个事件给 `ScoreBoard` 对象来更新 `plays` 变量的现实。当这个函数返回时，在 `runGame` 方法中的 `switch` 条件分支语句就会改变 `gameState` 为 `GameStates.STATE_START_REPLACING`，这样 `board` 才能用新的方块对象来重新填满自己。

```
public function removeClickedBlocks():void {
    removeClickedBlocksFromScreen();
    moveBlocksDown();
    clickedBlocksArray = new Array();
    dispatchEvent(new CustomEventScoreBoardUpdate(
        CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_PLAYS, String(plays)));
}
```

`removeClickedBlocksFromScreen` 方法是直线型执行的代码。它包括了一个 `while` 循环来从 `clickedBlocks` 数组中删除方块然后调用 `removeBlock` 方法，传递一个方块对象引用作为参数，这样这些方块就可以从屏幕上被移除了。它也会播放 `Main.SOUND_BONUS` 声效给玩家一个动作的声音反馈。

这个方法同时也包括了游戏得分部分的代码。我们创建一个局部变量叫做 `pointsPerBlock`，并且设置它的初始值为 `difficulty` 的值 `POINTS_PER_BLOCK`。在 `while` 循环内部，我们用 `BONUS_PER_BLOCK` 增加 `pointsPerBlock` 的值，把每一个连接的方块对象作为移除更多方块的奖励单位。我们把 `Math.floor(pointsPerBlock)` 得到的值当做参数传递给 `addToScore` 方法，`Math.floor(pointsPerBlock)` 表示最终产生的得分值的取整值。按四舍五入的话，玩家得到第五个方块处才能得到 1 点，到 5-9 个方块时才能得到 2 点，10 个或 10 个以上才能得到 3 点，以此类推。这种得分机制将鼓励玩家一次点击尽可能多的消除方块，因为得分会在一轮时间里飞快的上涨。

```
public function removeClickedBlocksFromScreen():void {
    var blockLength:int = clickedBlocksArray.length-1;
    var pointsPerBlock:Number = POINTS_PER_BLOCK;
    dispatchEvent( new CustomEventSound(
```



```
CustomEventSound.PLAY_SOUND,Main.SOUND_BONUS,false,1,0,1));
```

```
while(clickedBlocksArray.length > 0) {
    addToScore(Math.floor(pointsPerBlock));
    tempBlock = clickedBlocksArray.pop();
    removeBlock(tempBlock);
    pointsPerBlock += BONUS_PER_BLOCK;
}
}
```

addToScore 方法通过传递过来的参数增加 score 的值和 levelScore 的值，然后通过派发事件将值发送出去。

```
public function addToScore(val:Number):void {
    score += int(val);
    levelScore += int(val);
    dispatchEvent(new CustomEventScoreBoardUpdate(
        CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_SCORE,String(score
    )));
    dispatchEvent(new CustomEventScoreBoardUpdate(
        CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BOARD_LEVEL_SCORE,
        String(levelScore)));
}
```

moveBlocksDown 是游戏中复杂度仅次于 findLikeColoredBlocks 的方法。这个方法的工作就是定位通过 removeClickedBlocksFromScreen 方法消除的方块的位置并更新这些方块与其他在屏幕中方块的相对关系。这项工作涉及查找在 board 数组中消失的方块对象，并移动存在于屏幕上方块对象到被移除方块对象消失所产生的空位置，并标记自己的位置为 null 以使新的方块可以添加进来。

首先我们需要循环遍历 board 数组中的行和列，并且通过我们已掌握的更有效率的方式来操作它。因为方块落到了屏幕中，这意味着着屏幕网格中最低位的方块产生最短位移的方块，如果一个方块在这些方块的下面。也正因为这个原因，我们打算向后循环遍历 board 这个二维数组。这意味着我们首先要检测最右下角的方块。通过这个方向的检测，我们可以确定我们计数消失方块下方方块的数。我们不关心有多少方块在它们上方，因为这些上方的方块不影响新方块落到它们位置这个过程。这个小技巧帮助我们少写了很多代码，虽然是通过增加额外的循环遍历的开销。

我们首先通过反向循环遍历 board 数组中的列。对每一个列来说，我们在循环遍历每一个行。

```
public function moveBlocksDown():void {
    var collength:int = BLOCK_COLS-1;
    for (var c:int = collength; c >= 0; c--) {
        var rowlength:int = BLOCK_ROWS-1;
```



```
var missing:Number = 0;
for (var r:int = rowlength; r >= 0; r--) {
    tempBlock=board[r][c];
```

当我们发现一个行列组合不等于 null 时,我们跳到前面并检测在 board 二维数组中有多少消失的方块在它下面。我们通过创建另外一个 for 循环遍历所有 board[r][c]列在当前方块 (从 r+1 到 BLOCK\_ROWS-1) 下方来完成这个操作。如果我们发现了一个消失的方块,我们就使 missing 变量增加。

```
    if (tempBlock != null) {
        missing=0;
        if (r<BLOCK_ROWS) {
            for (var m:int = r+1; m < BLOCK_ROWS;m++) {
                if (board[m][c]==null) {
                    missing++;
                }
            }
        }
    }
}
```

在我们计算完有多少方块对象在通过 tempBlock 表示的当前列的下面时,是时候移动它了。首先,我们要改变方块的 row 属性和 col 属性来和新的方块位置吻合。这会在它完成落下动作之后。

```
    if (missing > 0) {
        tempBlock.row = r+missing;
        tempBlock.col = c;
```

然后,我们更新这个方块在 board 中新的位置,这个位置就是 tempBlock 现在逻辑上存在的位置,并且设置当前位置为 null 以便有新的方块来填补这个位置。

```
        board[r+missing][c] = tempBlock;
        board[r][c] = null;
```

最终,我们通过方块在屏幕中新的位置来调用 tempBlock 对象的 startFalling 方法,反过来说,也就是这个方块开始落下了。

```
        tempBlock.startFalling(tempBlock.y+
            (missing*BLOCK_HEIGHT)+(missing*ROW_SPACING),10);
    }
}
}
}
}
```



removeBlock 方法和我们之前在书中写的其他移除操作类的方法很相似。我们传递一个我们从 board 数组中移除的方块的引用给它。然后我们必须遍历数组知道我们定位到我们想要移除的那个方块位置。下一步工作就是尽可能多的清除这个方块后面的方块。我们移除它的事件监听器，通过调用 removeChild(tempBlock) 从显示列表里移除它，调用它的 dispose 方法(这将清除在内存中的位图数据)，并且设置它在 board 这个多维数组中的位置信息为 null。

```
public function removeBlock(blockToRemove:Block):void {
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempBlock = board[r][c];
            if (tempBlock == blockToRemove) {
                tempBlock.removeEventListener(
                    CustomEventClickBlock.EVENT_CLICK_BLOCK, blockClickListener);
                tempBlock.dispose();
                removeChild(tempBlock);
                board[r][c] = null;
            }
        }
    }
}
```

在图 8-6，当方块重新添加到 board 数组中在方块从屏幕中被移除时，你会发现它是怎样的。

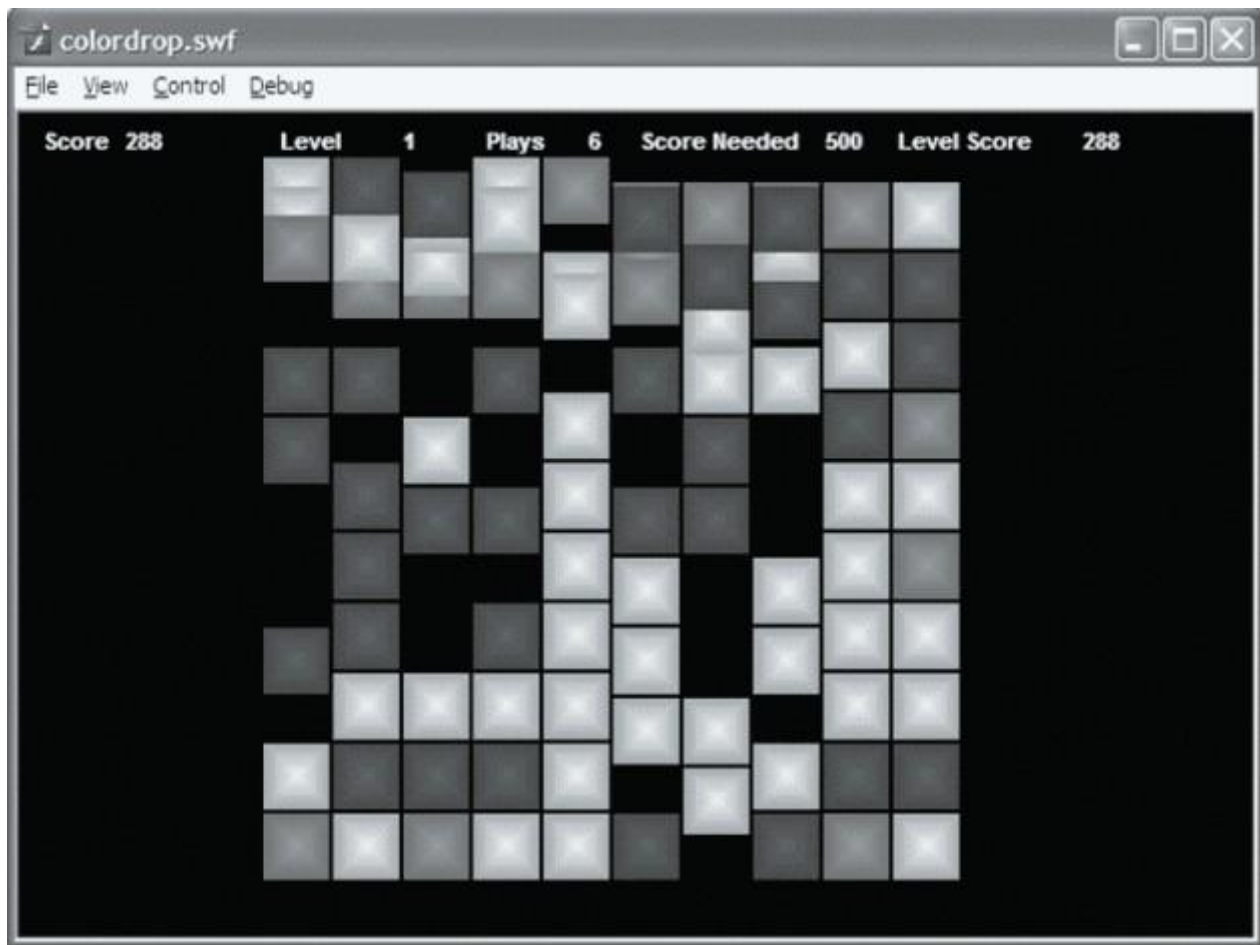


图 8-6. 方块重新被添加到屏幕中

## 结束当前等级或游戏

所以现在我们需要把方块放到屏幕中，清除它们，然后添加新的方块填补空白。现在唯一需要做的事情就是检测判断当前等级是否结束，或者游戏是否结束。我们在游戏状态为 `GameStates.STATE_WAITING_FOR_INPUT` 时通过调用 `checkForEndLevel` 方法来操作。

这个方法做了两件事，首先，它检测了 `levelScore` 是否大于等于 `currentLevel.scoreThreshold`。如果是，当前等级完成。我们播放一个声效，将 `nextGameState` 的值设定为 `GameStates.STATE_END_LEVEL`，调用 `fadeBlocks` 方法，且设置 `gameState` 的值为 `GameStates.STATE_FADE_BLOCKS_WAIT`。回想一下，在继续将 `gameState` 的值设定为 `nextGameState` 之前，`GameStates.STATE_FADE_BLOCKS_WAIT` 继续检测判断是否所有的方块都已经从屏幕中淡出。

如果等级还没有完成，我们检测判断 `plays` 是否已经耗尽。如果是，游戏结束了。我们进行类似的操作来结束当前级别。我们播放一个不同的声效；我们将 `GameStates.STATE_END_GAME` 值赋给 `nextGameState`，调用 `makeBlocksFall` 方法，将 `GameStates.STATE_FALL_BLOCKS_WAIT` 值赋给 `gameState`。

```
public function checkforEndLevel():void {
```



```
if (levelScore >= currentLevel.scoreThreshold) {
    dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND,
    Main.SOUND_WIN,false,1,0,1));
    nextGameState = GameStates.STATE_END_LEVEL;
    fadeBlocks();
    gameState = GameStates.STATE_FADE_BLOCKS_WAIT;
} else if (plays <= 0) {
    nextGameState = GameStates.STATE_END_GAME;
    dispatchEvent( new
CustomEventSound(CustomEventSound.PLAY_SOUND,Main.SOUND_LOSE,false,1,0,1));
    makeBlocksFall()
    gameState = GameStates.STATE_FALL_BLOCKS_WAIT;
}
}
```

fadeBlocks 方法循环遍历二维数组 board 中的所有方块 ,并且通过调用每一个方块的 startFade 方法来实现方块的淡出效果 ; 详见图 8-7。Block.startFade 方法接受一个 fadeValue ( 在 0 和 1 之间 ) 变量来决定方块经过几帧变化之后最终的 alpha 值。我们发送一个随机值使得方块的淡出不是完全统一的。

```
public function fadeBlocks():void {
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempBlock = board[r][c];
            tempBlock.startFade((Math.random()*9)+.1);
        }
    }
}
```

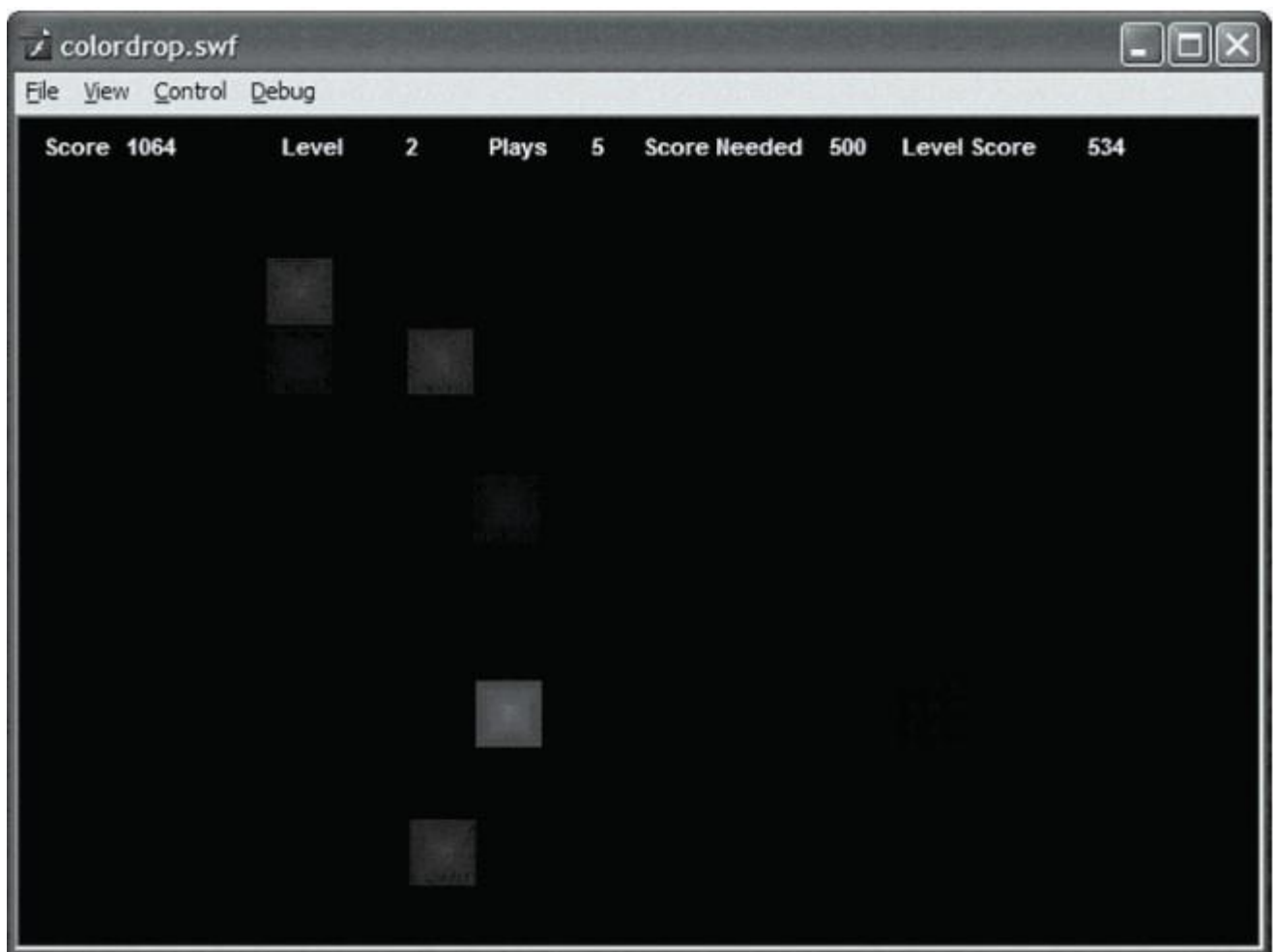


图 8-7. 方块淡出结束一个等级

makeBlocksFall 方法和 fadeBlocks 方法很相似。它循环遍历了在 board 二维数组中所有的方块对象。取代调用 startFade 方法，它调用了 startFalling 方法，传递方块的高度和屏幕底端的和作为第一参数（fallEndY）。我们传递这个值使得整个对象在从屏幕清除之前可以离开屏幕。第二个参数是一个随机值，这个值传递给方块的速度值（使得下落速度不是统一的）。

```
public function makeBlocksFall():void {  
    for (var r:int = 0; r < board.length; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempBlock = board[r][c];  
            tempBlock.startFalling(gameHeight + BLOCK_HEIGHT, (Math.random()*15)+10);  
        }  
    }  
}
```



最终，我们来看看 endGame 方法和 endLevel 方法。这两个方法在他们派发合适的事件给 Main 之前调用 cleanUpLevel 方法。cleanUpLevel 方法通过调用 removeBlock 方法循环遍历了 board 二维数组中所有的方块。

```
public function endGame():void {
    cleanUpLevel();
    dispatchEvent(new Event(GAME_OVER));
}

public function endLevel():void {
    cleanUpLevel();
    dispatchEvent(new Event(NEW_LEVEL));
}

public function cleanUpLevel():void {
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempBlock = board[r][c];
            removeBlock(tempBlock);
        }
    }
}
```

你也应该在 ColorDrop.as 类定义的括弧之内完成所有的操作：

```
}
}
```

## 测试！

好的，现在我们已经有了一个用 ActionScript 3.0 写的，使用可复用的游戏框架设计的益智休闲类游戏的全部代码。一旦你输入了所有的代码（或者下载它），测试一下游戏吧。注意到游戏开始的时候非常简单并且难度会越来越快的变高。通过在 DifficultyLevel 类里尝试使用不同的值，你也可以改变游戏的玩法和难度。

## 总结

在这一章节，我们已经建立了一个统一的益智类游戏基础代码，且我们在之前章节也讨论过。我们也为匹配网格中的物体对象建立了一个内部的游戏状态机和迭代解决方案。并且我们写的代码始终构建在我们之前已经建立的游戏框架的基础上。

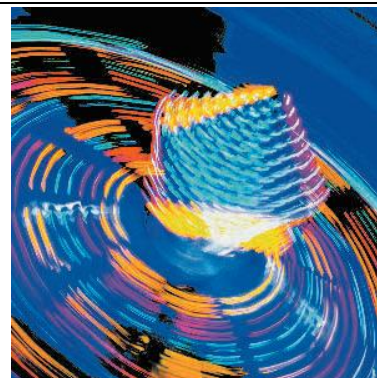


注意到，魔法色块这个游戏非常简单，后面的游戏和构思就不会这么简单了。它们几乎为任何你想创造的各种类型的游戏奠定了一定的基础。当然，使用迭代和递归方法检测方块在玩家点击之后可能会使游戏变成另外的样子，但是这个游戏为你探索其他游戏类型开辟了一条道路。

下面是一些你需要仔细考虑的问题：

- 怎么做才能使这个游戏可玩性更高，可玩时间更长？
- 怎样才能使游戏更容易玩？
- 该做什么样的改变才能使游戏不允许对角线的方块被匹配？
- 游戏中加入什么才能使游戏成为一个老少皆宜，爱不释手的游戏？

虽然我们完成了魔法色块游戏的代码，但并不是完成了这类游戏。下一章节，我们将使用在这个基础的引擎和改进它使它适应过去几年中发展出来的几种休闲游戏类型的方式继续探索更好的创意和想法。



## 第九章 骰子大战

在此之前的部分内容是有关美国版权的，不翻译。

### 设计骰子战斗游戏

对于骰子战斗游戏，如图9-1所示，我们可以利用 poker dice ,drop-style 类型的游戏像宝石迷阵以及我们自己的游戏

color drop，但是我们更愿意使用我们自己的设计制作一个游戏。在上一章节，我们讨论过休闲游戏的历史，但是我们

遗忘了match-three style 风格的游戏像宝石迷阵。那个游戏如此的流行以至于许多其他风格的游戏模仿它，

利用它的基本玩法。其中一种风格就是智能对战。然而在一个更大的环境中探索，

像贪吃虫大冒险和益智之谜之类已经采用了drop-style类型的智力游戏的思想。

我们将要制作的游戏 骰子大战 是在color drop上添加一些特色让它更加接近智能对战游戏。

实际上，我们没有检查互联网上成千上万的游戏是否已经存在这种类型的游戏。不管怎么样，我们



这里的意图

是让我们的游戏在一些基本的思想上添加一些属于我们自己的想法。如图9-1

**游戏名字：**骰子大战

**游戏风格：**Drop-style 智能战斗游戏

**游戏描述：**轮流移动格子中的骰子通过连续的敌人

**玩家目标：**尽可能多的连接相同点数 相同颜色的骰子然后消除。这些点数将伤害一个玩家，

这些点数等于从敌人身上移除的伤害值。

**敌人描述：**敌人是玩这游戏越来越好的一系列生物

**敌人目标：**敌人试图尽可能的从玩家身上移除伤害值。最终目的是从玩家身上移除所有的伤害值。

**等级结束：**敌人丢失所有的伤害值

**游戏结束条件：**玩家丢失所有伤害值

**难度系数：**难度按特定的等级增加。敌人的智能也

**奖励条件：**与消去的骰子目数与个数有关

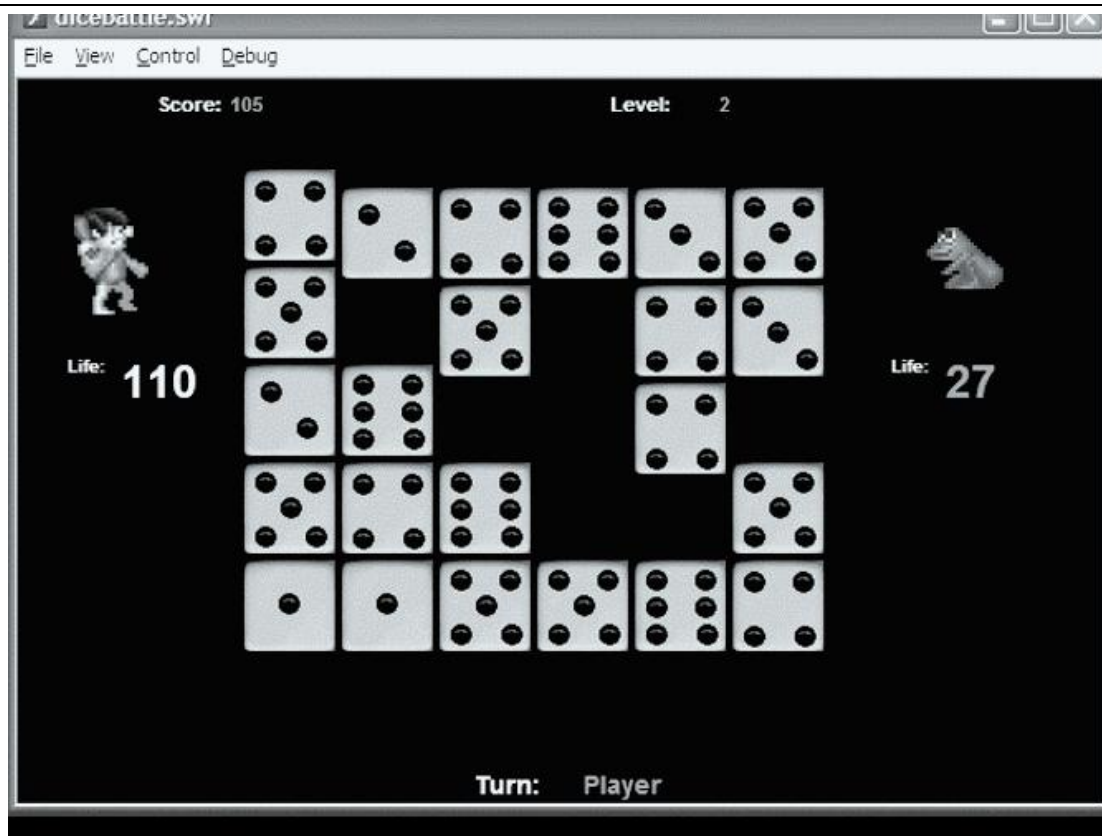


图 9-1

## 本章游戏理念

骰子大战是重复上一章Color Drop。尽管在这一章我们重新创建了所有的代码，然而我们参考了Color Drop和第8章的

许多场景。骰子大战我们可以理解为一个独立的游戏，然而我们强烈建议您在阅读第9章的内容之前通读第8章而且编译Color Drop。

在创建骰子大战的同时，我们将介绍一下游戏开发与设计的主题



在现有的游戏中创建另一个

播放声音

创建计算机对手

轮流

AI 基础算法

对于骰子大战，我们将创建以下类：

DiceBattle：游戏主类，继承之com.efg.framework.Game

DifficultyLevel：难度等级类。这个类规定游戏的难度等级，骰子大战有10个难度等级。

Character：角色类 这个类继承之com.efg.framework.BlitSprite 用于在屏幕上显示玩家和计算机角色

Die：死亡类。我们将使用这个类表现屏幕上的每一次消除。这个类很像上一章中的block类除了Die类同时包含了颜色和值。

CustomEventClickDie:玩家事件类。玩家点击消除后将发送者个事件。

GameStates：游戏状态类。这个类保存所有的静态常量，这些静态常量为骰子大战的状态机描述游戏内部状态

## 添加游戏资源到库中

和Color Drop 不同，骰子大战只需要非常少的资源。这些资源包括大量的图形和额外的几个声音。

详细描述如下：



## 添加图形到骰子大战

对于骰子大战，我们需要2个不同的遮盖层。第一层是我们将在游戏中使用的不同颜色的骰子。

随着游戏难度等级的提升，我们将向屏幕中添加不同颜色的骰子。只有相同颜色的骰子可以匹配，所以随着颜色增多，很难从这么多的骰子中找出匹配的。我们可以在代码中用三个设置来标记骰子的颜色以解决我们碰到的问题，你也可以自己创建这些图形以达到和代码相同的相同的效果，但是这个好像不是总是那么容易。

每次消除是50\*50px 如图9-2

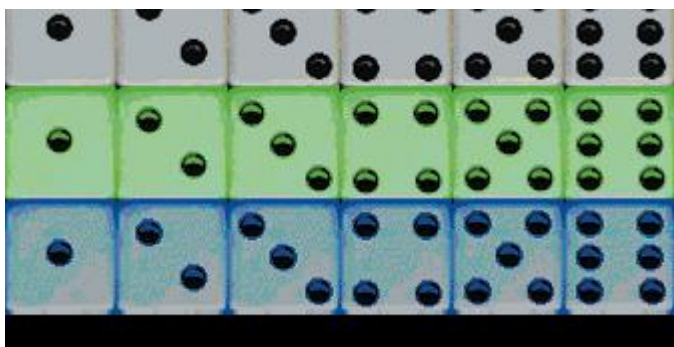


图9-2

第二个覆盖层是游戏中的角色设置。第一个角色是玩家。随后的所有角色都是敌人。那个玩家将在游戏中战斗。

我们创建10个敌人，所以在游戏中有10个等级。和这本书中其他图形一样，这些角色图片来自Ari Feldman's SpriteLib GPL

每个角色是64px\*64px 如图9-3 这一层的库名叫：Characters

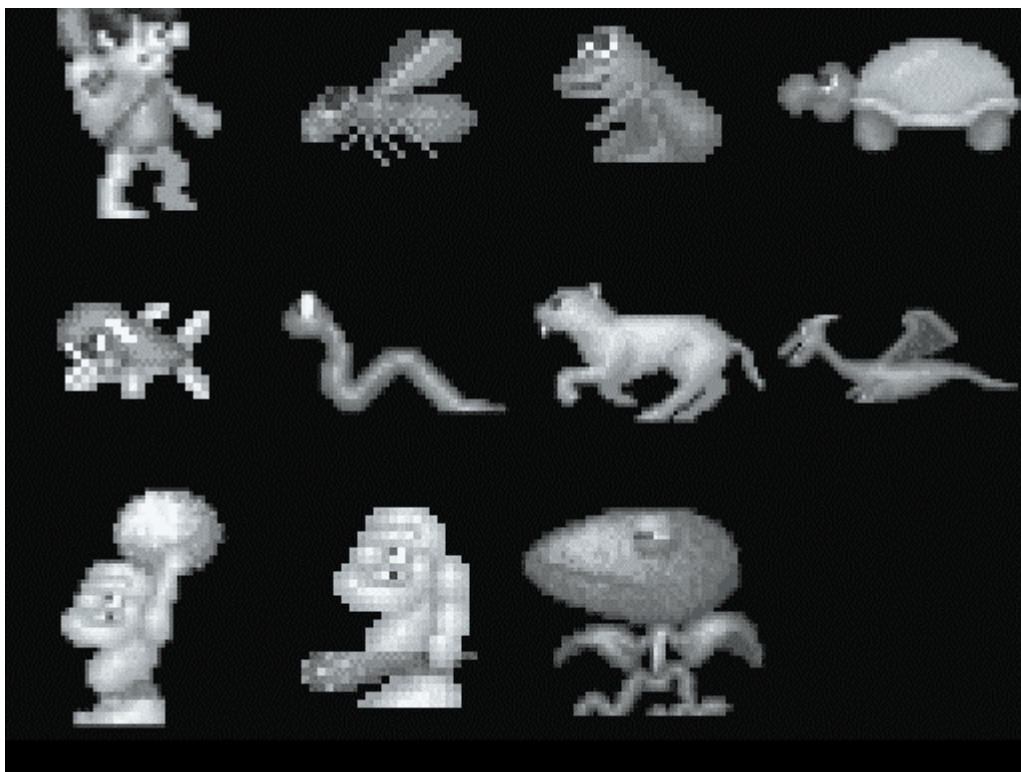


图9-3

## 给骰子大战添加声音

骰子大战大多数声音和Color Drop相同。然而，我们添加了3个新的。当轮到计算机移动时播放打击音。

这个声音让人感觉起来像玩家正在痛击头部。另外两个声音是SoundTrack1 和SoundTrack2。这2个声音做为配乐在游戏中循环播放。考虑到配乐比较特殊，在稍后的部分，我们将向你介绍一种技巧使用多重声道在游戏中播放不同次数的声音。最终完整的声音清单见表 9-1

**表9-1 骰子大战的声音**

类名	描述
SoundBonus	玩家连接骰子
SoundClick	玩家点击骰子
SoundWin	玩家胜利



SoundLose	玩家失败
SoundHit	计算机连接骰子
SoundTrack1	屏幕配乐 ?
SoundTrack2	游戏主配乐 ?

## 在 Main.as 文件中播放配乐

我们首先讨论的大部分代码在Main.as中。骰子大战的Main.as 类中的代码大部分和Color Drop是相同的。

不过，这里主要有2个不同的地方：记分板和配乐播放。

## 修改记分板

对于骰子大战来说，我们将要创建一个比之前我们创建过的更加复杂的记分板。尽管这些代码并不复杂诡异，但我们任然要探讨它。因为它能够被用于比当前我们创建的更加复杂的设计中。注意骰子大战的游戏屏幕 如图 9-4 我们高亮了5个记分元素。

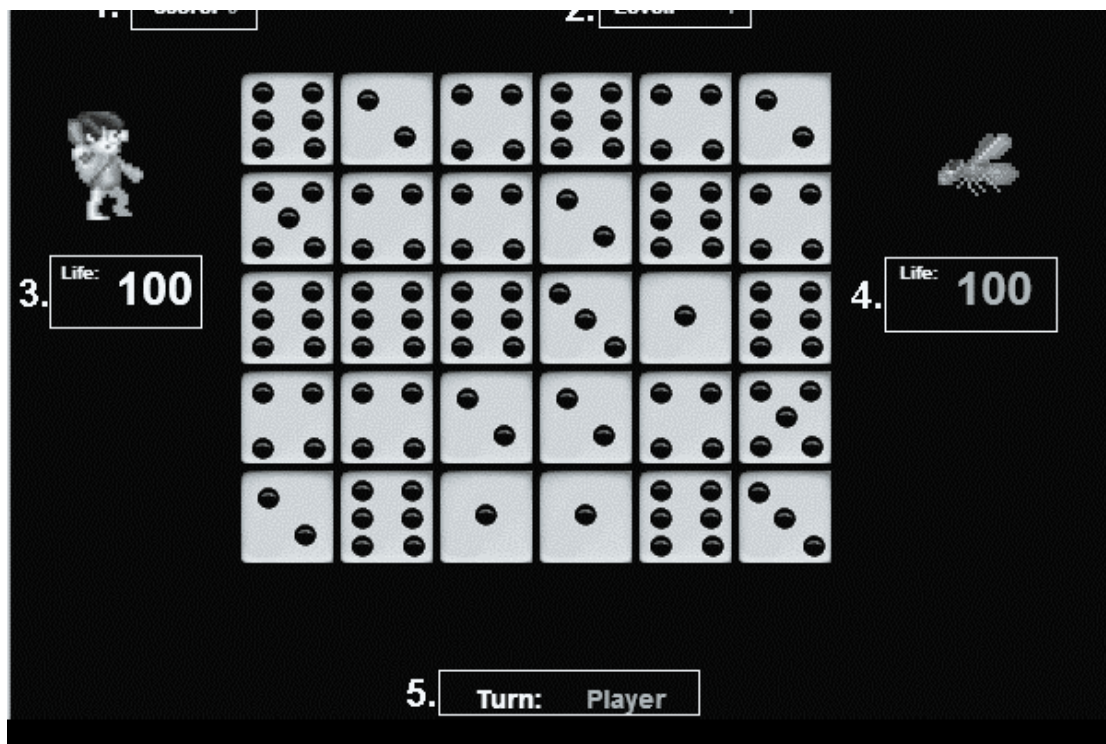


图 9-4 骰子大战屏幕的记分板



这里是骰子大战的记分元素

玩家的分数

当前游戏等级

玩家的生命值

计算机生命值

轮次标记

所有的元素必须让玩家知道游戏中正在发生着什么。为了显示这些元素我们需要创建几个 文本对象。

```
var scoreBoardTextFormat1:TextFormat = new TextFormat("_sans", "11", "0xffffffff", "true");
var scoreBoardTextFormat2:TextFormat = new TextFormat("_sans", "11", "0xff0000", "true");
var scoreBoardTextFormat3:TextFormat = new TextFormat("_sans", "14", "0xffffffff", "true");
var scoreBoardTextFormat4:TextFormat = new TextFormat("_sans", "14", "0xff0000", "true");
var scoreBoardTextFormat5:TextFormat = new TextFormat("_sans", "10", "0xffffffff", "true");
var scoreBoardTextFormat6:TextFormat = new TextFormat("_sans", "25", "0x00ff00", "true");
var scoreBoardTextFormat7:TextFormat = new TextFormat("_sans", "25", "0xff0000", "true");
scoreBoard.createTextElement(SCORE_BOARD_SCORE, new SideBySideScoreElement(□
75, 5, 15, "Score:", scoreBoardTextFormat1, 25, "0", scoreBoardTextFormat2));
scoreBoard.createTextElement(SCORE_BOARD_LEVEL, new SideBySideScoreElement(□
325, 5, 10, "Level:", scoreBoardTextFormat1, 50, "0", scoreBoardTextFormat2));
scoreBoard.createTextElement(SCORE_BOARD_TURN, new SideBySideScoreElement(□
250, 380, 10, "Turn:", scoreBoardTextFormat3, 50, "0", scoreBoardTextFormat4));
scoreBoard.createTextElement(SCORE_BOARD_PLAYER_LIFE, new SideBySideScoreElement(□
25, 150, 10, "Life:", scoreBoardTextFormat5, 20, "0", scoreBoardTextFormat6));
scoreBoard.createTextElement(SCORE_BOARD_COMPUTER_LIFE, new SideBySideScoreElement(□
480, 150, 10, "Life:", scoreBoardTextFormat5, 20, "0", scoreBoardTextFormat7));
```

## 播放配乐

在第四章，我们创建了能够播放配乐(背景音乐)的声音管理类。在许多智能对战风格的游戏，像骰子大战，都包含了精心制作的配乐用以设置当前游戏状态的情绪。不管怎么样，当你使用状态机去运行你的



游戏时,管理不同的配乐是个不错的技巧。在main类中 我们使用2个静态常量SOUND\_SOUND\_TRACK\_1和SOUND\_SOUND\_TRACK\_2 定义这2种不同的配乐:

```
public static const SOUND_SOUND_TRACK_1:String = "soundTrack1";
public static const SOUND_SOUND_TRACK_2:String = "soundTrack2";
...
soundManager.addSound(SOUND_SOUND_TRACK_1, new SoundTrack1());
soundManager.addSound(SOUND_SOUND_TRACK_2, new SoundTrack2());
```

我们可以在各种游戏状态播放这些配乐,见表:9-2

**表 9-2 Main.as中的配乐**

状态描述	状态函数	播放配乐
屏幕标题显示	systemTitle	SoundTrack1
开始新游戏	systemNewGame	SoundTrack2
游戏结束	systemGameOver	SoundTrack2
屏幕标题显示	systemTitle	SoundTrack1

*注意systemTitle 方法将在 systemGameOver方法之后调用,然而他们却播放相同的配乐。*

现在,这就是为什么我们需要在SoundManager类中创建一个特殊的方法来播放配乐。因为在GameFrameWork(Main继承之这个类)类中游戏的各个状态时独立的,systemTitle方法肯定不会知道systemGameOver是游戏的最后状态。事实上,这里可能会有一些其他的状态在中间(比如:在屏幕上不断显示一个高分)。

然而,这2个状态都会播放配乐SoundTrack1,因此我们为配乐的playSoundTrack方法创建一个布尔型的参数isSoundTrack。这个方法被设计成一次只播放一个配乐。回想第4章,我是通过创建一个独立的SoundChannel类来播放这些配乐的。这个类不仅播放配乐,而且阻止状态机过多的了解游戏状态流转的细节,降低模块之间的耦合度。



为了在Main类中播放配乐,我们需要覆写systemTitle,systemNewGame和systemGameOver方法。

每一个新方法直接调用soundManager.playSound方法,然后再通过super.[方法名]调用父类的同名方法。

Main类修改如下:

```
override public function systemTitle():void {
    soundManager.playSound(Main.SOUND_SOUND_TRACK_1,true,1000,0,1);
    super.systemTitle();
}
override public function systemNewGame():void {
    soundManager.playSound(Main.SOUND_SOUND_TRACK_2,true,1000,0,1);
    super.systemNewGame();
}
override public function systemGameOver():void {
    soundManager.playSound(Main.SOUND_SOUND_TRACK_1,true,1000,0,1);
    super.systemGameOver();
}
```

## 更新 Main.as 类

下面是Main类的完整代码清单.我们在这个版本中使用粗体标明了修改的地方:

```
package com.efg.games.dicebattle
{
    import com.efg.framework.FrameWorkStates;
    import com.efg.framework.GameFrameWork;
    import com.efg.framework.BasicScreen;
    import com.efg.framework.ScoreBoard;
    import com.efg.framework.Game;
    import com.efg.framework.SideBySideScoreElement;
    import com.efg.framework.SoundManager;
    import com.efg.framework.CustomEventSound;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.utils.Timer;

    import flash.events.TimerEvent;
```



```
import flash.text.TextFormat;

public class Main extends GameFrameWork
{
    //custom score board elements
    //屏幕显示元素
    public static const SCORE_BOARD_SCORE:String = "score";
    public static const SCORE_BOARD_LEVEL:String = "level";
    public static const SCORE_BOARD_TURN:String = "turn";
    public static const SCORE_BOARD_COMPUTER_LIFE:String = "computerLife";
    public static const SCORE_BOARD_PLAYER_LIFE:String = "playerLife";
    //custom sounds
    //声音
    public static const SOUND_CLICK:String = "soundClick";
    public static const SOUND_BONUS:String = "soundBonus";
    public static const SOUND_WIN:String = "soundWin";
    public static const SOUND_LOSE:String = "soundLose";
    public static const SOUND_HIT:String = "soundHit";
    public static const SOUND_SOUND_TRACK_1:String = "soundTrack1";
    public static const SOUND_SOUND_TRACK_2:String = "soundTrack2";
    /**Flex Framework Only
    /*以下仅在flex框架中使用
    [Embed(source = "assets/dicebattleassets.swf", symbol="SoundClick")]
    private var SoundClick:Class;
    [Embed(source = "assets/dicebattleassets.swf", symbol="SoundBonus")]
    private var SoundBonus:Class;
    [Embed(source = "assets/dicebattleassets.swf", symbol="SoundWin")]
    private var SoundWin:Class;
    [Embed(source = "assets/dicebattleassets.swf", symbol="SoundLose")]
    private var SoundLose:Class; //chnaged
    [Embed(source = "assets/dicebattleassets.swf", symbol="SoundHit")]
    private var SoundHit:Class;
    [Embed(source = "assets/dicebattleassets.swf", symbol="SoundTrack1")]
    private var SoundTrack1:Class;
    [Embed(source = "assets/dicebattleassets.swf", symbol="SoundTrack2")]
    private var SoundTrack2:Class;
    */
    // Our construction only calls init(). This way, we can re-init the
    entire system if necessary
    //构造函数中只调用init(), 这种方式让我们可以重新初始化整个系统。
    public function Main() {
        init();
    }
    // init() is used to set up all of the things that we should only
```



```
need to do one time
//init() 用于初始化所有东西
override public function init():void {
    game = new DiceBattle(600,400);
    setApplicationBackGround(600,400,false, 0x000000);
    //add application background to the screen as the bottom layer
    //添加背景到屏幕做为最底层
    //add score board to the screen as the seconf layer
    //添加记分板到屏幕作为第2层
    scoreBoard = new ScoreBoard();
    addChild(scoreBoard);
    var scoreBoardTextFormat1:TextFormat = new TextFormat("_sans", "11", "0xffffffff", "true");
    var scoreBoardTextFormat2:TextFormat = new TextFormat("_sans", "11", "0xff0000", "true");
    var scoreBoardTextFormat3:TextFormat = new TextFormat("_sans", "14", "0xffffffff", "true");
    var scoreBoardTextFormat4:TextFormat = new TextFormat("_sans", "14", "0xff0000", "true");
    var scoreBoardTextFormat5:TextFormat = new TextFormat("_sans", "10", "0xffffffff", "true");
    var scoreBoardTextFormat6:TextFormat = new TextFormat("_sans", "25", "0x00ff00", "true");
    var scoreBoardTextFormat7:TextFormat = new TextFormat("_sans", "25", "0xff0000", "true");
    scoreBoard.createTextElement(SCORE_BOARD_SCORE, □
    new SideBySideScoreElement(75, 5, 15, "Score:", □
    scoreBoardTextFormat1, 25, "0", scoreBoardTextFormat2));
    scoreBoard.createTextElement(SCORE_BOARD_LEVEL, □
    new SideBySideScoreElement(325, 5, 10, "Level:", □
    scoreBoardTextFormat1, 50, "0", scoreBoardTextFormat2));
    scoreBoard.createTextElement(SCORE_BOARD_TURN, □
    new SideBySideScoreElement(250, 380, 10, "Turn:", □
    scoreBoardTextFormat3, 50, "0", scoreBoardTextFormat4));
    scoreBoard.createTextElement(SCORE_BOARD_PLAYER_LIFE, □
    new SideBySideScoreElement(25, 150, 10, "Life:", □
    scoreBoardTextFormat5, 20, "0", scoreBoardTextFormat6));
    scoreBoard.createTextElement(SCORE_BOARD_COMPUTER_LIFE, □
    new SideBySideScoreElement(480, 150, 10, "Life:", □
    scoreBoardTextFormat5, 20, "0", scoreBoardTextFormat7));
    //screen text initializations
    //屏幕文本初始化
    screenTextFormat = new TextFormat("_sans", "14", "0xffffffff", "true");
    screenButtonFormat = new TextFormat("_sans", "11", "0x000000", "true");
    titleScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE, □
    600,400, false, 0x000000);
    titleScreen.createDisplayText("Dice Battle! ", 120, new Point(255, □
    150), screenTextFormat);
    titleScreen.createOkButton("Go!", new Point(250, 250), 100, 20, □
    screenButtonFormat, 0xFFFFFFFF, 0x00FF0000, 2);
```



```
instructionsScreen = new BasicScreen(□
FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS, 600,400, false, 0x000000);
instructionsScreen.createDisplayText("Beat The Computer!", □
250,new Point(230, 150),screenTextFormat);
instructionsScreen.createOkButton("Play", new Point(250,250), 100, 20, □
screenButtonFormat, 0xFFFFFFFF, 0xFF0000, 2);
gameOverScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER, □
600,400, false, 0x000000);
gameOverScreen.createDisplayText("Game Over", 300,new Point(250,100), □
screenTextFormat);
gameOverScreen.createOkButton("Play Again", new Point(225,250), 150, 20, □
screenButtonFormat, 0xFFFFFFFF, 0xFF0000, 2);
levelInText = "Level ";
levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER, □
600,400, false, 0x000000);
levelInScreen.createDisplayText(levelInText, 300,new Point(275,100), □
screenTextFormat);
//set initial game state
//游戏状态初始化
switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
waitTime= 40;
soundManager.addSound(SOUND_CLICK, new SoundClick());
soundManager.addSound(SOUND_BONUS, new SoundBonus());
soundManager.addSound(SOUND_WIN, new SoundWin());
soundManager.addSound(SOUND_LOSE, new SoundLose());
soundManager.addSound(SOUND_HIT, new SoundHit());
soundManager.addSound(SOUND_SOUND_TRACK_1, new SoundTrack1());
soundManager.addSound(SOUND_SOUND_TRACK_2, new SoundTrack2());
frameRate = 40;
startTimer();
}
override public function systemTitle():void {
    soundManager.playSound(Main.SOUND_SOUND_TRACK_1, true, 1000, 0, 1);
    super.systemTitle();
}
override public function systemNewGame():void {
    soundManager.playSound(Main.SOUND_SOUND_TRACK_2, true, 1000, 0, 1);
    super.systemNewGame();
}
override public function systemGameOver():void {
    soundManager.playSound(Main.SOUND_SOUND_TRACK_1, true, 1000, 0, 1);
    super.systemGameOver();
}
```



```
}  
}
```

## 创建 AI 难度类

回想上一章节，我们创建了一个难度等级(DifficultyLevel)类。这里我们将创建为骰子大战创建一个新的版本的。尽管我对用颜色来表示每一个难度等级很满意，但是我还是需要表示每一个级别的电脑对手的AI。这个类没有方法，我们将简单的介绍这些属性是如何与游戏联系起来的。

allowedColors：一个Die.DIE\_COLOR\_xxx 的数组 颜色值表示等级，这个游戏只有3个不同的颜色值。

enemyLife：表示敌人生命的一个值。当敌人的生命掉光后，玩家将赢得这一局。

aiBonus：这是一个几率，当我们解决移动时计算机有一点几率使敌人的AI上升一个等级

minValue：每次移动的最低值。帮助我们逐级增加游戏的难度。

enemyTile：我们将在屏幕中显示的游戏等级

以下是 DifficultyLevel 类的是完整的代码：

```
package com.efg.games.dicebattle  
{  
    public class DifficultyLevel  
    {  
        public var allowedColors:Array;  
        public var enemyLife:Number;  
        public var aiBonus:Number;  
        public var minValue:Number;  
        public var enemyTile:Number  
        public function DifficultyLevel(allowedColors:Array, enemyLife: Number,   
        enemyTile:Number, aiBonus:Number, minValue:Number) {  
            this.allowedColors = allowedColors;  
            this.enemyLife = enemyLife;  
            this.aiBonus = aiBonus;  
            this.enemyTile = enemyTile;  
            this.minValue = minValue;  
        }  
    }  
}
```



}

## 创建死亡类

Die 类与 Color Drop 中的 Block 类很接近。这里我们将使用 Block 类作为 Die 类的基类。我们将高亮与 Block 类中不同的部分，我们必须明白我们现在使用的是骰子而已不是简单的彩色方块。

显然，彩色骰子与彩色方块最大的不同之处在于骰子有一个值。这个值用来表示面对着die。热心的玩家应该已经知道这些骰子可能有4-20个面或者更多。然而，对于这个游戏，我们只需要标准的6面体骰子，值的范围从1-6.为了在die类中表示着些值，我们将创建一个number类型的公共的属性dieValue;我们也需要为骰子创建颜色值。在骰子大战中将有3种颜色的骰子：

```
public static var DIE_COLOR_WHITE:int = 0;
public static var DIE_COLOR_GREEN:int = 1;
public static var DIE_COLOR_BLUE:int = 2;
```

在die类的构造函数中，我们需要初始化dieValue并找到合适的tile来显示die.第一部分是很容的，我们只需要把参数dieValue赋给this.dieValue即可：

然而找到合适的face tile 在tile sheet 有一点点棘手。代码如下：

```
var tile:int = (dieValue-1) + (dieColor)*6;
```

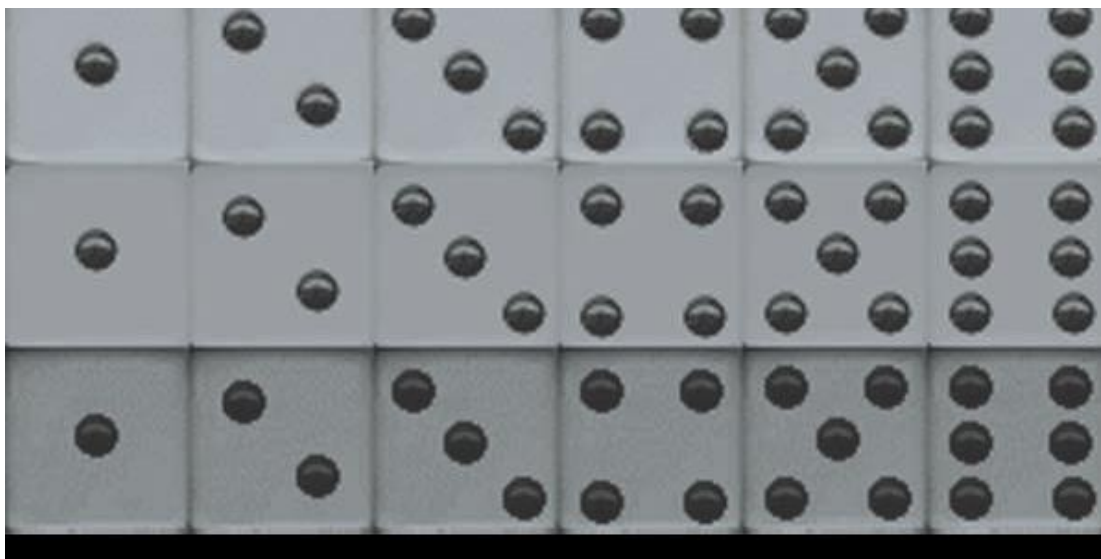
这段代码需要简单说明。我们知道 die类的构造函数初始化时会给dieColor和dieValue赋值。

注意图9-5 tilesheet中的骰子按数值顺序一行一种颜色排列的。我们要做的是通过tile来计算出骰子所在的行列位置。Tiles是一个基于0的数组。这就意味着我们必须从dieValue中减掉1。因此第一部分的计算是：dieValue-1.

下一步，我们需要找到当前颜色所在的行。由于颜色的静态常量值是从0开始的，如果我们加上dieColor\*6，我们将找到这正确的行。在我们的例子中，dieValue等于5，dieColor等于1.。将这些值代入到等式

$((dieValue-1) + (dieColor)*6)$ 中可以得到 $(5 - 1) + (1 * 6) = 10$ .因为tiles是基于0的数组，所以被选

中的tile是第十一个，见图9-5.



**图9-5 被选中的tile高亮显示**

对于block类来说我们在die类中增加了一个更重要的东西，见下面的2个方法：

```
public function makeDieClickedComputer() : void {  
    this.filters=[new flash.filters.GlowFilter(0xFF0000, 70, 4, 4, 3, 3, false, false)];  
}
```

注意，这2个方法是很接近的(颜色不同)。

```
public function makeDieClicked() : void {  
    this.filters=[new flash.filters.GlowFilter(0xFFFF00, 70, 4, 4, 3, 3, false, false)];  
}
```

当骰子被点击的时候，makeDieClickedComputer方法只是简单的使骰子由黄色变为红色。这里我们  
可以通过使用一个方法改变颜色的值来代替2个方法，但是我们认为这个例子有助于我们说明计算机的AI  
比优化代码要多一些。

```
package com.efg.games.dicebattle  
{  
    import flash.filters.GlowFilter;  
    import flash.events.MouseEvent;  
    import flash.display.Bitmap;  
    import flash.display.BitmapData;  
    import com.efg.framework.BlitSprite;  
    import com.efg.framework.TileSheet;
```



```
public class Die extends BlitSprite {
    public var dieColor:Number;
    public var dieValue:Number;
    //Action vars
    public var isFalling:Boolean;
    public var isFading:Boolean;
    public var fadeValue:Number = .05;
    public var fallEndY:Number;
    public var speed:int = 10;
    public var nextYLocation:int = 0;
    //Board Info
    public var row:Number;
    public var col:Number;
    public static var DIE_COLOR_WHITE:int = 0;
    public static var DIE_COLOR_GREEN:int = 1;
    public static var DIE_COLOR_BLUE:int = 2;
    public function Die(dieValue:Number, dieColor:Number, tileSheet:TileSheet, □
    row:Number, col:Number, endY:Number, speed:Number) {
        this.dieColor = dieColor;
        this.dieValue = dieValue;
        this.row = row;
        this.col = col;
        isFalling = false;
        isFading = false;
        var tile:int = (dieValue-1) + (dieColor)*6;
        super(tileSheet, [tile], 0);
        this.addEventListener(MouseEvent.CLICK, onMouseDownListener, false, 0, true);
        this.buttonMode = true;
        this.useHandCursor = true;
        startFalling(endY, speed);
    }
    public function startFalling(endY:Number, speed:Number) : void{
        this.speed = speed;
        fallEndY = endY;
        isFalling=true;
    }
    public function startFade(fadeValue:Number) : void {
        this.fadeValue = fadeValue;
        isFading=true;
    }
    public function update() : void {
        if (isFalling) {
            nextYLocation = y + speed;
```



```
}  
}  
public function render() : void {  
    if (isFalling) {  
        y=nextYLocation;  
        if (y >= fallEndY) {  
            y = fallEndY;  
            isFalling=false;  
        }  
    }  
    if (isFading) {  
        alpha -= fadeValue;  
        if (alpha <=0) {  
            alpha = 0;  
            isFading = false;  
        }  
    }  
}  
public function onMouseDownListener(e:MouseEvent) : void {  
    dispatchEvent(new CustomEventClickDie(CustomEventClickDie.EVENT_CLICK_DIE, this));  
}  
public function makeDieClicked() : void {  
    this.filters=[new flash.filters.GlowFilter(0xFFFFF00, 70, 4, 4, 3, 3, false, false)];  
}  
public function makeDieClickedComputer() :void {  
    this.filters=[new flash.filters.GlowFilter(0xFF0000, 70, 4, 4, 3, 3, false, false)];  
}  
}  
}
```

顺便说一下，做下小许改变，我们可以采用 oop(面向对象方式)思想让 block 作为基类，die 继承之。

不过，这个我们留给你自己去探索。

## 创建 CustomEventClickDie 类

CustomEventClickDie 类只是在第8章Color Drop的CustomEventClickBlock 类上做少部分修改。

理论上，我们可以重用CustomEventClickBlock类或者创建一个可重用的类加入到

com.efg.framework包中。



不过，这里我们选择重写这个类，因为我们想加强你在游戏中创建各种自定义事件的意识。尽管游戏框架是一个很有用的工具，但是它不会为你写游戏。这个类唯一真正的区别是使用die类代替block类做为这个类的唯一变量的类型。

```
package com.efg.games.dicebattle
{
    import flash.events.*;
    public class CustomEventClickDie extends Event
    {
        public var die:Die;
        public static const EVENT_CLICK_DIE:String = "eventClickDie";
        public function CustomEventClickDie(type:String, die:Die, bubbles:Boolean=false, cancelable:Boolean=false)
        {
            super(type, bubbles, cancelable);
            this.die = die;
        }
        public override function clone():Event {
            return new CustomEventClickDie(type, die, bubbles, cancelable)
        }
        public override function toString():String {
            return formatToString(type, "type", "bubbles", "cancelable", "eventPhase");
        }
    }
}
```

## 创建角色类

角色类是一个非常简单的类。我将使用这个类来象征库中的tile。从某种意义上说，这个类是BlitSprite的一个子集。我们创建这个类是为了让它在将来游戏升级时能够扩展。在学习完这章之后，你应该随时保持这种思想，“我能否使用自己的想法和代码来改进这个游戏。”

```
package com.efg.games.dicebattle
{
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
```



```
import com.efg.framework.BlitSprite;
import com.efg.framework.TileSheet;
public class Character extends BlitSprite {
    public function Character(ts:TileSheet, tile:Number) {
        super(ts, [tile], 0);
    }
}
}
```

## 添加游戏状态类

为了让计算机玩家有AI，我们需要为骰子大战创建一些新的游戏状态。在第9章，我们介绍了游戏状态机概念：a substate machine used inside the game class independent of the one in Main. 我们创建游戏状态类做为一个独立的类是为了在ColorDrop的代码中更容易控制和关联。我们将再次使用这些思想在骰子大战上。然而，对于骰子大战来说，我们需要扩展状态机，为Color Drop的状态机添加2个额外的状态，

我们需要的这个 2 个状态是 STATE\_CHANGE\_TURN(决定是电脑还是玩家移动)和 STATE\_START\_AI(创建一个计算机移动的进程)。剩下的新状态指示名字与 Color Drop 中的不同。我们使用 dice 代替 block。

```
package com.efg.games.dicebattle
{
    public class GameStates {
        public static const STATE_INITIALIZING:int = 10;
        public static const STATE_CHANGE_TURN:int = 20;
        public static const STATE_START_REPLACING:int = 30;
        public static const STATE_START_AI:int = 40;
        public static const STATE_WAITING_FOR_INPUT:int = 60;
        public static const STATE_REMOVE_CLICKED_DICE:int = 70;
        public static const STATE_CHECK_FOR_END:int = 80;
        public static const STATE_FADE_DICE_WAIT:int = 90;
        public static const STATE_FALL_DICE_WAIT:int = 100;
        public static const STATE_END_GAME:int = 110;
        public static const STATE_END_LEVEL:int = 120;
    }
}
```



```
public static const STATE_WAIT:int = 130;
}
}
```

## 在 game.as 中初始化设置

现在，我们将开始对比Color Drop与骰子大战的game类,找出需要修改的地方。第一个我们需要修改的地方是game类的属性和构造函数。现在我们将通过高亮修改的地方来完成game类的第一部分。

事实上,这个类开始的部分和ColorDrop.as很像，直到具体的代码确定board将如何显示为止，我们发现没有任何改变。我们总共有30(5行6列)个die对象，这比在Color Drop中的block对象少的多。除此之外，

```
package com.efg.games.dicebattle
{
import flash.display.Sprite;
import flash.events.*;
import com.efg.framework.Game;
import com.efg.framework.CustomEventLevelScreenUpdate;
import com.efg.framework.CustomEventScoreBoardUpdate;
import com.efg.framework.CustomEventSound;
import com.efg.framework.TileSheet;
public class DiceBattle extends Game
{
private var gameWidth:int;
private var gameHeight:int;
private static const Y_PAD:int = 75;
private static const X_PAD:int = 150;
private static const ROW_SPACING:int = 4;
private static const COL_SPACING:int = 4;
private static const DIE_HEIGHT:int = 50;
private static const DIE_WIDTH:int = 50;
private static const DIE_ROWS:int = 5;
private static const DIE_COLS:int = 6;
```

这个类中第一个重要的不同是HAR\_WIDTH 和CHAR\_HEIGHT。这2个变量确定了我们将角色tile sheet 中裁剪的 角色tiles的大小，我们将在init函数中使用这个角色tile。接下来的2个新变量我们用来保存玩家和电脑的生命值。我们通过自定义事件正确的显示这2个值在在记分板上。Turn变量保存当前的次序



是玩家还是计算机。这些值保存在2个静态常量 TURN\_PLAYER 和 TURN\_COMPUTER 中。

```
private static const CHAR_WIDTH:int = 64;
private static const CHAR_HEIGHT:int = 64;
private var score:int;
private var level:int;
private var playerLife:int;
private var computerLife:int;
private var turn:String;
private static const TURN_PLAYER:String = "Player's";
private static const TURN_COMPUTER:String = "Computer's";
private var difficultyLevelArray:Array;
private var currentLevel:DifficultyLevel;
```

接下来的新变量控制游戏的难度的设置。这些变量也可以作为难度设置，但是，既然我们已经为这个游戏创建了DifficultyLevel类，那么这样做也就没有必要了。 playerLifeStart参数每一个等级开始时玩家的初始化生命值（附近上一等级剩余的生命值）。 DICE\_BONUS是玩家或者计算机玩家在单轮中连接一个die(结束一个)获得的奖励点数。另外的新变量只是简单的改下名字：clickedBlockArray改成 clickedDiceArray ，

tempBlock改成tempDie 。

```
private var playerLifeStart:int = 100;
private const DICE_BONUS:int = 2;
private var clickedDiceArray:Array;
private var board:Array;
private var tempDie:Die;
private var gameState:int = 0;
private var nextGameState:int = 0;
private var framesToWait:int = 0;
private var framesWaited:int = 0;
```

顺便说一下，你没有想过，从这个游戏和 Color Drop 中提炼出一些代码做为一个基类在这种类型的其他游戏中使用？你打算怎么做？仔细思考这个问题在你阅读这一章的时候。

在这个游戏中我们有2个tile sheets可以使用，DiceSheet 和角色。下面的代码创建一些变量来保存它



们。

我们也创建了2个变量来保存我们之前确定的角色实例，一个是玩家角色(playerTile)另一个是计算机角色。

```
private var tileSheet:TileSheet;
private var charSheet:TileSheet;
private var enemyTile:Character;
private var playerTile:Character;
//***** Flex *****
//[Embed(source = 'assets/dicebattleassets.swf', symbol = 'Characters')]
//private var Characters:Class;
//[Embed(source = 'assets/dicebattleassets.swf', symbol = 'DiceSheet')]
//private var DiceSheet:Class;
//***** End Flex *****
```

骰子大战的构造函数和 Color Drop是很接近的，代码如下：

```
public function DiceBattle(gameWidth:int, gameHeight:int) {
    this.gameWidth=gameWidth;
    this.gameHeight=gameHeight;
    init();
    gameState = GameStates.STATE_INITIALIZING;
}
```

骰子大战的init函数与上一个游戏我们创建的init函数非常类似：它初始化tile sheets和难度等级。而且初始化了2个tiles sheets(DiceSheet 和Characters)，这里几乎没有新的东西需要探讨。但是，这里的DifficultyLevel类有点意思。下面是一个我们将要创建的DifficultyLevel类的例子

```
difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE], 150, 3, 20, 7));
```

现在，让我们回忆一下传递给DifficultyLevel类的参数，因为我们很快将使用它们。

[Die.DIE\_COLOR\_WHITE]：我们将在这一等级使用的骰子的颜色数组

150：敌人的生命值

3：这一等级显示的敌人角色的tile

20：AI奖励附加到从0-99生成的随机值 来帮助计算机决定如何移动。



7：敌人可接受的最小移动值。（如果有值等于或大于它）

我们采用相同的形式为这个游戏创建了 10 个等级。为了增加游戏的难度，我们可以增加骰子的颜色来使匹配更难，增加敌人的生命值，提高 AI 奖励和降低分数门槛。所有的这些操作能够也应该保持游戏的平衡。

因为游戏牵涉到 AI，所以我们必须真正的撕破这个游戏如何使用这些值的表面。你必须去试验，看看一个小小的改变是如何影响整个游戏的难度的。

```
public function init():void
{
    //***** Flash IDE *****
    tileSheet = new TileSheet(new DiceSheet(0,0), DIE_WIDTH,DIE_HEIGHT);
    charSheet = new TileSheet(new Characters(0,0), CHAR_WIDTH,CHAR_HEIGHT);
    //***** End Flash IDE *****
    //***** Flex *****
    //tileSheet = new TileSheet(new DiceSheet().bitmapData, DIE_WIDTH, DIE_HEIGHT);
    //charSheet = new TileSheet(new Characters().bitmapData, CHAR_WIDTH, CHAR_HEIGHT);
    //***** End Flex ***** difficultyLevelArray = new Array();
    difficultyLevelArray = new Array();
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE],100,1,0,0));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE],125,2,10,6));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE],150,3,20,7));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE, □
    Die.DIE_COLOR_BLUE],175,4,30,7));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE, □
    Die.DIE_COLOR_BLUE],200,5,40,7));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE, □
    Die.DIE_COLOR_BLUE],225,6,50,7));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE, □
    Die.DIE_COLOR_BLUE],250,7,60,7));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE, □
    Die.DIE_COLOR_BLUE,Die.DIE_COLOR_GREEN],275,8,70,7));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE, □
    Die.DIE_COLOR_BLUE,Die.DIE_COLOR_GREEN],300,9,80,7));
    difficultyLevelArray.push(new DifficultyLevel([Die.DIE_COLOR_WHITE, □
    Die.DIE_COLOR_BLUE,Die.DIE_COLOR_GREEN],325,10,90,7));
}
```

The newGame function does very little new in Dice Battle except initialize both playerLife and



computerLife. These values will be set in newLevel.

newGame 函数除了初始化playerLife 和computerLife之外做了很少新的东西。在每一个新的等级这些值都会被重新初始化。

```
override public function newGame():void {  
    level = 0;  
    score = 0;  
    playerLife=0;  
    computerLife=0;  
}
```

## 创建计算机玩家

DiceBattle类与ColorDrop类最大的变动是DiceBattle类必须创建一个智能的计算机玩家与人类玩家

对战。这些变动开始产生严重的情况在newLevel方法中。

```
override public function newLevel():void  
{  
    level++  
    var tempLevel:int = level;  
    if (tempLevel > (difficultyLevelArray.length-1)) {  
        tempLevel = difficultyLevelArray.length-1  
    }  
    currentLevel = difficultyLevelArray[tempLevel-1];
```

我们需要做的第一件事就是估算奖励生命。奖励生命是在游戏开始一个新的级别时附加给玩家的生命值。

对于这个游戏，玩家从上一级别剩余生命值中获得10%附加到 playerStartLife中。我们也设置计算机的生命，计算机的生命值保存在DifficultyLevel类的实例currentLevel中。

```
var bonusLife:int = (Math.ceil(playerLife/10));  
playerLife = playerLifeStart + bonusLife;  
computerLife = currentLevel.enemyLife;  
turn = "";  
board = new Array();  
for (var r:int = 0; r < DIE_ROWS; r++) {  
    board[r] = new Array();  
    for (var c:int = 0; c < DIE_COLS; c++) {  
        board[r][c] = null;  
    }  
}
```



```
}
```

现在，我们需要取得enemyTile和playerTile然后在屏幕上显示这些图形。要做到这一点，为其各自创建新的角色类并传递我们想要附加的相关参数。

```
clickedDiceArray = new Array();
enemyTile = new Character(charSheet, currentLevel.enemyTile);
playerTile = new Character(charSheet, 0);
playerTile.x = 50;
playerTile.y = 100;
enemyTile.x = 525;
enemyTile.y = 100;
this.addChild(playerTile);
this.addChild(enemyTile);
```

最后，我们发送事件更新记分板。playerLife和computerLife是需要我们跟踪的2个新的值，在新的级别开始时我们需要设置它们。同样我们需要决定轮到谁来移动，我们只需要设置GameStates.STATE\_CHANGE\_TURN即可。GameStates.STATE\_CHANGE\_TURN将决定下一轮由哪一个玩家(人类或计算机)来移动。

```
dispatchEvent(new CustomEventLevelScreenUpdate(□
CustomEventLevelScreenUpdate.UPDATE_TEXT, String(level)));
dispatchEvent(new CustomEventScoreBoardUpdate(□
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_SCORE, String(score)));
dispatchEvent(new CustomEventScoreBoardUpdate(□
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_LEVEL, String(level)));
dispatchEvent(new CustomEventScoreBoardUpdate(□
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_COMPUTER_LIFE, □
String(computerLife)));
dispatchEvent(new CustomEventScoreBoardUpdate(□
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_PLAYER_LIFE, □
String(playerLife)));
gameState = GameStates.STATE_CHANGE_TURN;
}
```

## 使用新的游戏状态来轮流移动

骰子大战的runGame方法已经说明了两个玩家（人类玩家和计算机玩家）在游戏中轮流移动。虽然这



不是一个很卖座的2人游戏，但是只要做小许改变它能够很容易成为。我们将简短的说明一下主要的修改

在你阅读完下面的代码之后：

```
override public function runGame():void
{
    switch(gameState)
    {
        case GameStates.STATE_INITIALIZING:
            break;
```

新的参数gameState ( GameStates.STATE\_CHANGE\_TURN ) 在newLevel方法中初始化。由于turn参数在级别开始时设置为空,所以第一轮总会是玩家先移动。从那时起，我们简单的转换TURN\_PLAYER和TURN\_COMPUTER的值直到这一级别结束或者game over。我们也发送一个事件给scoreboard来更新屏幕最底部的次序文本。

```
case GameStates.STATE_CHANGE_TURN:
    if (turn == TURN_COMPUTER || turn == "") {
        turn = TURN_PLAYER;
    } else {
        turn = TURN_COMPUTER
    }
    dispatchEvent(new CustomEventScoreBoardUpdate(□
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_TURN, String(turn)));
    gameState = GameStates.STATE_START_REPLACING;
    break;
```

当新的骰子被替换后，我们检查此时是谁的轮。如果是玩家我们简单的侦听鼠标输入通过设置gameState为GameStates.STATE\_WAITING\_FOR\_INPUT.如果是计算机，我们需要创建一个AI移动，通过设置gameState为GameStates.STATE\_START\_AI。然后我们设置gameState为GameStates.STATE\_FALL\_DICE\_WAIT来等待被替换的骰子落入这个位置，在Color Drop中我做了相同的事。

```
case GameStates.STATE_START_REPLACING:
    replaceDice();
    if (turn == TURN_PLAYER) {
```



```
nextGameState = GameStates.STATE_WAITING_FOR_INPUT;
} else {
nextGameState = GameStates.STATE_START_AI;
}
gameState = GameStates.STATE_FALL_DICE_WAIT;
break;
case GameStates.STATE_WAITING_FOR_INPUT:
break;
```

接下来的 gameState 是 GameStates.STATE\_START\_AI。这个状态本质上是产生计算机玩家移动。为了产生这个移动，我们调用 createAIMove 方法。

```
case GameStates.STATE_START_AI:
createAIMove();
framesToWait = 15;
framesWaited = 0;
nextGameState = GameStates.STATE_REMOVE_CLICKED_DICE;
gameState = GameStates.STATE_WAIT;
break;
case GameStates.STATE_REMOVE_CLICKED_DICE:
removeClickedDice();
gameState = GameStates.STATE_CHECK_FOR_END;
break;
```

STATE\_CHECK\_FOR\_END state. The rest of the states remain essentially unchanged from Color Drop.

在接下来的代码中，检查等级或者游戏是否结束是有一点点复杂的。由于在玩家或者计算机完成一轮之前我们是不知道游戏或者等级是否已经结束。因此我们需要在这个GameStates.

STATE\_CHECK\_FOR\_END状态中做两种检查。从Color Drop保留的剩余的状态本质上没有改变。

```
case GameStates.STATE_CHECK_FOR_END:
if (checkforEndLevel()) {
nextGameState = GameStates.STATE_END_LEVEL;
fadeDice();
gameState = GameStates.STATE_FADE_DICE_WAIT;
} else if (checkForEndGame()) {
nextGameState = GameStates.STATE_END_GAME;
```



```
makeDiceFall()
gameState = GameStates.STATE_FALL_DICE_WAIT;
} else {
gameState= GameStates.STATE_CHANGE_TURN;
}
break;
case GameStates.STATE_FADE_DICE_WAIT:
if (!checkForFadingDice()) {
gameState = nextGameState;
}
break;
case GameStates.STATE_FALL_DICE_WAIT:
if (!checkForFallingDice()) {
gameState = nextGameState;
}
break;
case GameStates.STATE_END_LEVEL:
endLevel();
gameState = GameStates.STATE_INITIALIZING;
break;
case GameStates.STATE_END_GAME:
endGame();
break;
case GameStates.STATE_WAIT:
framesWaited++;
if (framesWaited >= framesToWait) {
gameState = nextGameState;
}
break;
}
update();
render();
}
```

## 玩家移动占领和得分

dieClickListener方法中的代码几乎和Color Drop中的dieClickListener函数没有两样，事实上只是把每一行代码中的“block”单词替换成“die”或者“dice”。一个很小但是很重要的修改是这个方法接收CustomEventClickDie的实例代替CustomEventClickBlock。然而，与Color Drop中的



removeClickedBlocks 方法最大的不同是我们用来计算得分的方法removeClickedDice，因为我们使用removeClickedDice计算2个玩家的移动。通过移动来计算得分，removeClickedDice方法能够保持一个独立的职责而不用知道谁调用了它。最重要的代码我们已经附加在下面。首先，我们取得dice值通过传递一个clickedDiceArray参数给getDiceValue方法。然后我们调用addToScore方法。

```
var dicevalue:Number = getDiceValue(clickedDiceArray);
dispatchEvent(new CustomEventScoreBoardUpdate(□
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_COMPUTER_LIFE, □
String(computerLife)));
addToScore(dicevalue);
```

以下是dieClickListener方法的完整代码：

```
public function dieClickListener(e:CustomEventClickDie):void {
    if (gameState == GameStates.STATE_WAITING_FOR_INPUT) {
        dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND, □
Main.SOUND_CLICK, false, 1, 0, 1));
        tempDie = e.die;
        clickedDiceArray = findLikeColoredDice(tempDie);
        for (var i:int =0; i< clickedDiceArray.length; i++) { //changed dice battle
            clickedDiceArray[i].makeDieClicked(); //changed dice battle
        }
        var dicevalue:Number = getDiceValue(clickedDiceArray);
        dispatchEvent(new CustomEventScoreBoardUpdate(□
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_COMPUTER_LIFE, □
String(computerLife)));
        addToScore(dicevalue);
        framesToWait=15;
        framesWaited=0;
        dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND, □
Main.SOUND_BONUS, false, 1, 0, 1));
        nextGameState = GameStates.STATE_REMOVE_CLICKED_DICE;
        gameState = GameStates.STATE_WAIT;
    }
}
```

findLikeColoredDice方法与Color Dro中的findLikeColoredBlocks方法很像。最主要的不同是我们需



要考虑每个Die类的实例的diceValue和diceColor当我们测试匹配的时候。为了做到这一点，我们将在两个地方执行检查：

```
if (tempDie.dieColor == tColor && tempDie.dieValue == tValue)
```

基本上，我们检查dieColor 和 dieValue二者就能从die对象中找到匹配的。如果有匹配，我们将这个die对象添加到diceMatched数组中。想完全的理解代码是如何工作的，请参考第8章的findLikeColoredBlocks的相关讨论。

```
private function findLikeColoredDice(tDie:Die):Array {
    var diceToCheck:Array = new Array();
    var diceMatched:Array = new Array();
    var diceTested:Array = new Array();
    var rowList:Array = [-1, 0, 1, -1, 1, -1, 0, 1];
    var colList:Array = [-1, -1, -1, 0, 0, 1, 1, 1];
    var tColor:Number = tDie.dieColor;
    var tValue:Number = tDie.dieValue;
    diceToCheck.push(tDie);
    while(diceToCheck.length > 0) {
        tempDie = diceToCheck.pop();
        if(tempDie.dieColor == tColor && tempDie.dieValue == tValue){ //changed for dice drop
            diceMatched.push(tempDie);
        }
        var tB2:Die;
        for (var i:int = 0; i < rowList.length; i++) {
            if ((tempDie.row + rowList[i]) >= 0 && (tempDie.row + rowList[i]) < DIE_ROWS && 
                (tempDie.col + colList[i]) >= 0 && (tempDie.col + colList[i]) < DIE_COLS) {
                var tr:int = tempDie.row + rowList[i];
                var tc:int = tempDie.col + colList[i];
                tB2 = board[tr][tc];
                if(tB2.dieColor == tColor && tB2.dieValue == tValue && diceToCheck.indexOf(tB2) != 
                    -1 && diceTested.indexOf(tB2) == -1){
                    diceToCheck.push(tB2)
                }
            }
        }
        diceTested.push(tempDie);
    }
}
```



```
return diceMatched;
}
```

getDiceValue是一个全新的方法。这个方法对于ColorDrop来说是没有必要的，因为那个游戏中魔法色块没有值。这个方法接受一个die对象作为参数。这个方法遍历数组并且计算所有dice的值，数组中第一个之后的每一个die的bonusPoints(奖励值)通过DICE\_BONUS递增。增加奖励值意味着相同的移动的中包含的骰子越多获得的奖励也就越多。

```
public function getDiceValue(tdice:Array):Number {
    var value:int = 0;
    var bonusPoints:int = 0;
    for (var d:int = 0;d< tdice.length;d++)
    {
        tempDie = tdice[d];
        value += (tempDie.dieValue+bonusPoints);
        bonusPoints += DICE_BONUS;
    }
    return value;
}
```

addToScore方法与Color Drop中相同的方法是很接近的。主要的不同是computerLife被消耗通过总分(从计算机身上移除的生命值)，而且会在记分板上显示。

```
private function addToScore(val:Number):void {
    score += int(val);
    computerLife -= int(val);
    dispatchEvent(new CustomEventScoreBoardUpdate(□
    CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_SCORE, String(score)));
    dispatchEvent(new CustomEventScoreBoardUpdate(□
    CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_COMPUTER_LIFE, □
    String(computerLife)));
}
```



## 创建计算机 minimax-style 风格的 AI

当游戏的状态被设置为GameStates.STATE\_START\_AI时会调用createAIMove方法。

createAIMove方法的职责是模拟玩家的移动和得分让我们以为是一个玩家在移动。

AI的移动与玩家的移动不同的是AI移动将减少playerLife值代替减少computerLife的值，由此来降低玩家的生命值。

当我们试图弄清楚怎样为这个游戏创建AI时，一个想法呈现在我们的脑海里：计算机与玩家的移动没有什么不同，仅仅是我们得到的结果不同。

此外，如果我们模拟一个玩家的行动来选择合适的die来消除，我们将得到更多的点数，事实上，我们能够创建一个人类对战的AI。

这种类型的AI被称为miniMax，这种是真正的“min or max.”。这种风格的AI在AI玩家拥有完整的信息并且能够计算出最大优化移动和最小优化移动然后在这些信息的基础上决定如何移动对于游戏来说是很重要的。

这种风格的AI通常牵涉到一个判断树，它包含了整个游戏所有可能的移动，因此使得AI玩家可以计算出最好的移动在任何时候直到得到最终结果。

然而，在我们的游戏中使用这种风格的AI暂时还有一个问题。AI玩家仅仅拥有骰子显示在当前板面上的完整信息。

它没有移动骰子和补充骰子产生的信息。这里有一些局限存在因为我们产生的新的骰子拥有随机的颜色和值。

因此，看来在当前任何时候我们仅仅只能生成一份AI移动的清单。这就意味着我们的minimax AI仅仅拥有一堆我们能够通过value来排序的移动。



同样，我们按一定的比例来提高AI，这样做的目的是计算机不会总是做最好的移动，以此方式让我们得到了游戏的难度等级。

这就是为什么我们叫它 minimax-style AI：

## 分析骰子的 AI

来想象一下计算机玩家在骰子大战中是如何产生AI的，我们首先坐下来讨论下计算机需要知道的关于游戏的一些东西：

轮到计算机时，它必须进行移动。

能够获得所有可能的移动在它的一轮中。

每一步移动的值（从玩玩家身上移除的生命值）

能够选择一个最好的移动

能够产生一个最差的移动。

接着，我们一次性解决上面所有的问题。

第一个问题，让计算机知道进行一个移动是非常容易解决的。在之前的规定中，我们创建turn变量和 GameStates.STATE\_CHANGE\_TURN 游戏状态来解决这个问题。基本上，在计算机要做决定的时候将调用 createAIMove 方法。

下一个问题，计算机能够在它的一轮里能够获得所有可能的移动是有一点点难度的。我们已经有了一个 findLikeColoredDice 接受一个 die 的实例参数。这个方法返回一个具有相同颜色和值的 die 类的数组。通过创建这些指定的投资(移动的)的集合，计算机能够在任何一轮中知道每一个移动。

接下来的问题是计算机能够获得这些移动的骰子的值后与之前的结合起来。遍历所有可能移动的集合，我能够得到它们的值。我们也可以通过将这些移动排序来找出哪一个是最值得，第2个值得，第3个等等。

最后剩下的 2 个问题，计算机知道选择最好的移动来避免最差的移动，是通过 DifficultyLevel 类的属



性

aiBonus 和 minValue 来实现的。以上问题都是理论上的解答，让我们进入到实际编码中可能会更好的理解一些。

## 探讨骰子大战的 AI

为了更好的理解计算机AI如何移动，请看图9-6。从屏幕上来看，你可能已经知道轮到计算机移动，玩家的等级已经到达3级。

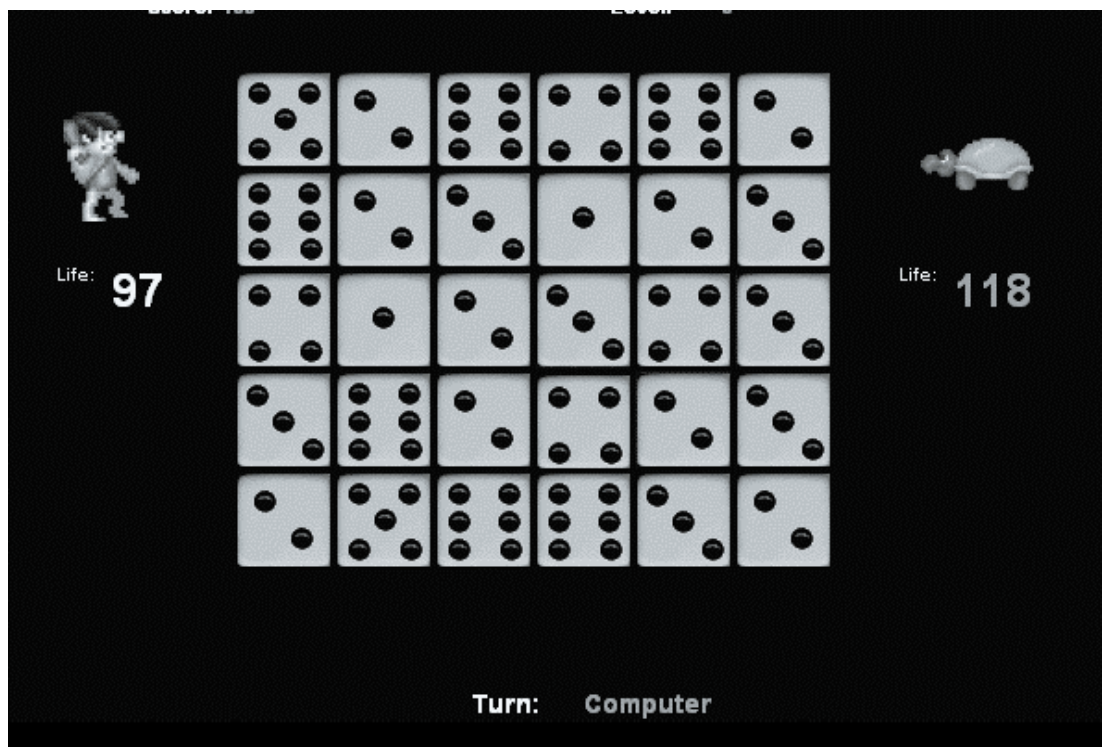


图 9-6 当轮到计算机移动的时候，计算机进行智能移动。

当轮到计算机移动时会调用createAIMove方法，下面的代码将被执行。

首先，我们找到计算机所有可能的移动。



我们首先通过2层遍历板面上的所有die对象。

我们取出每一个die并且调用findLikeColoredDice(tempDie)方法找到已经匹配的die对象。

我们认为这些是可移动的，因为如果我们假定计算机点击了这个tempDie，计算机将会试着移动一组从所有的die对象返回的die。这些移动路线只中每个die只出现一次。我们做这个测试之前我们投入到findLikecoloredDice方法的代码中去。

```
if (diceTested.indexOf(tempDie) == -1) {
```

Array.indexOf方法将检查diceTested数组中是否已经存在这个die.如果存在，将不需要测试。例如，如果一个3挨着一个另一个3，那么我们只需要将第一个3加入到可移动的结果中，因为点击这第2个3将产生一个相同的移动路线。这样做，我们可以减少需要我们计算和处理的移动路线的数量。

```
public function createAllMove():void {  
    var dieSets:Array = new Array();  
    var diceTested:Array = new Array();  
    var testSet:Array = new Array();  
    var testValue:Number = 0;  
    for (var r:int = 0; r < board.length; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempDie = board[r][c];  
            if (diceTested.indexOf(tempDie) == -1) {  
                testSet = findLikeColoredDice(tempDie);  
                testValue = getDiceValue(testSet);  
                dieSets.push([Number(testValue), tempDie]);  
                while(testSet.length > 0) {  
                    diceTested.push(testSet.pop());  
                }  
            }  
        }  
    }  
}
```



dieSets数组被创建为一个多维数组。第一部分是可移动的骰子的值。第2部分是我们将要移动的die对象。

如果我输出每一个结果中的第一个die的值，行，列，我们将得到下面的清单：

```
Moves:18
Value:5 Row:0 Col:0
Value:20 Row:3 Col:2
Value:6 Row:0 Col:2
Value:4 Row:0 Col:3
Value:6 Row:0 Col:4
Value:6 Row:1 Col:4
Value:6 Row:1 Col:0
Value:8 Row:2 Col:3
Value:1 Row:1 Col:3
Value:24 Row:4 Col:4
Value:4 Row:2 Col:0
Value:1 Row:2 Col:1
Value:10 Row:3 Col:3
Value:3 Row:3 Col:0
Value:24 Row:4 Col:3
Value:6 Row:4 Col:5
Value:2 Row:4 Col:0
Value:5 Row:4 Col:1
```

现在，我从低到高排列所有的可移动的骰子。最大值的，最好的先移动。采用数组排序来决定计算机应该移动哪一个对我们来说是很简单的事。我们关心的是，这可以移动的骰子的值，因此我们能够使用sortOn方法让这些可以移动的骰子按升序排序。

```
dieSets.sortOn("0", Array.NUMERIC);
```

以下是排序后的dieSets数组：

```
Sorted Moves: 18
Value:1 Row:1 Col:3
Value:1 Row:2 Col:1
```



---

```
Value:2 Row:4 Col:0
Value:3 Row:3 Col:0
Value:4 Row:2 Col:0
Value:4 Row:0 Col:3
Value:5 Row:0 Col:0
Value:5 Row:4 Col:1
Value:6 Row:1 Col:0
Value:6 Row:0 Col:2
Value:6 Row:4 Col:5
Value:6 Row:1 Col:4
Value:6 Row:0 Col:4
Value:8 Row:2 Col:3
Value:10 Row:3 Col:3
Value:20 Row:3 Col:2
Value:24 Row:4 Col:3
Value:24 Row:4 Col:4
```

接下来，我们从可移动的骰子列表中剥离value相同的让我们的计算更正容易点。执行这个操作是很容易的。

我们创建一个lastValue变量来保存骰子的值，在循环遍历可移动的骰子的值时 如果当前的骰子的值等于

lastValue 那么我们从数组中删除当前元素。

```
var lastValue:Number = 0;
for (var ii:int = dieSets.length-1; ii >= 0; ii--) {
    if (dieSets[ii][0] == lastValue) {
        dieSets.splice(ii, 1);
    } else {
        lastValue = dieSets[ii][0];
    }
}
```

以下是我们移除了所有相同的值后输出的dieSets数组：

```
No Like Values Moves:10
Value:1 Row:2 Col:1
Value:2 Row:4 Col:0
Value:3 Row:3 Col:0
```



```
Value:4 Row:0 Col:3
Value:5 Row:4 Col:1
Value:6 Row:0 Col:4
Value:8 Row:2 Col:3
Value:10 Row:3 Col:3
Value:20 Row:3 Col:2
Value:24 Row:4 Col:4
```

最后，我们移除每一个值小于当前等级设置中minValue的骰子()。对于等级3来说，

currentLevel.minValue

是7。这就意味着计算机总是选择得分大于等于7的移动，我们将留下4种移动供计算机选择：

```
if (dieSets[dieSets.length-1][0] > currentLevel.minValue) {
for (ii = dieSets.length-1; ii >= 0; ii--) {
if (dieSets[ii][0] < currentLevel.minValue) {
dieSets.splice(ii, 1);
}
}
}
```

以下是我们移除了值少于minValue的后输出的dieSets数组：

```
minValue removed Moves:4
Value:8 Row:2 Col:3
Value:10 Row:3 Col:3
Value:20 Row:3 Col:2
Value:24 Row:4 Col:4
```

现在我们从0-99中取得一个随机数。我们测试看看如果dieSets>1，因为我们不需要解决只有一个时将怎么移动。

```
var aiVal:Number = 0;
if (dieSets.length > 1)
{
var pChance:Number = Math.floor(Math.random() * 100);
```

以下是pChance输出的结果：

```
pChance without aiBonus:28
```



我们现在加上当前等级的aiBonus。值越大，计算机移动的越好。等级3的aiBonus默认值是20，所以我们计算后得到pChance等于48

```
pChance += currentLevel.aiBonus;
```

以下是加上aiBonus后pChance的值：

pChance with Albonus:48

现在，我们简单的使用99除以可移动骰子数(99/4)=24。我们保存24到pVal作为每一个可移动骰子的乘数。

接着我们用pChance除以pVal得到计算机将要使用的骰子保存到aiVal中。在这里，aiVal是2。然而，由于是一个索引基于0开头的数组，我们已经选择的骰子对于计算机来说在数组中的索引将是1。

```
var pVal:Number = Math.floor(99/dieSets.length);  
aiVal = Math.floor(pChance/pVal);  
if (aiVal > dieSets.length-1) {  
    aiVal = dieSets.length-1;  
}  
} else {  
    aiVal = 0;  
}
```

以下是计算完成后输出的aiVal的值

Chosen dieSet:1

输出的结果意味着计算机使用的移动是dieSets[1]，dieSets[1]的值是10.这表示将从玩家身上移除10点生命值。

```
tempDie = dieSets[aiVal][1];
```

createAIMove方法剩余要做的是模拟计算机真正做了移动。你注意到这些代码是如此的熟悉。这是因为它和玩家移动没有什么区别的。主要的不同是从playerLife中减掉部分代替从computerLife中并且在记分板中显示。并且播放不同的声音，调用die类中的方法makeDieClickedComputer。



```
clickedDiceArray = findLikeColoredDice(tempDie);  
for (var cd:int=0; cd< clickedDiceArray.length; cd++) {  
    clickedDiceArray[cd].makeDieClickedComputer();  
}  
playerLife -= dieSets[aiVal][0];  
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, Main.SOUND_HIT, □  
false, 1, 0, 1));  
dispatchEvent(new CustomEventScoreBoardUpdate(□  
CustomEventScoreBoardUpdate.UPDATE_TEXT, Main.SCORE_BOARD_PLAYER_LIFE, □  
String(playerLife)));  
}
```

图 9-7 显示当计算机移动时屏幕如何表现。注意红色的范围来是计算机选择的骰子。那个由调用 `clickedDiceArray[cd].makeDiceClickedComputer` 造成。

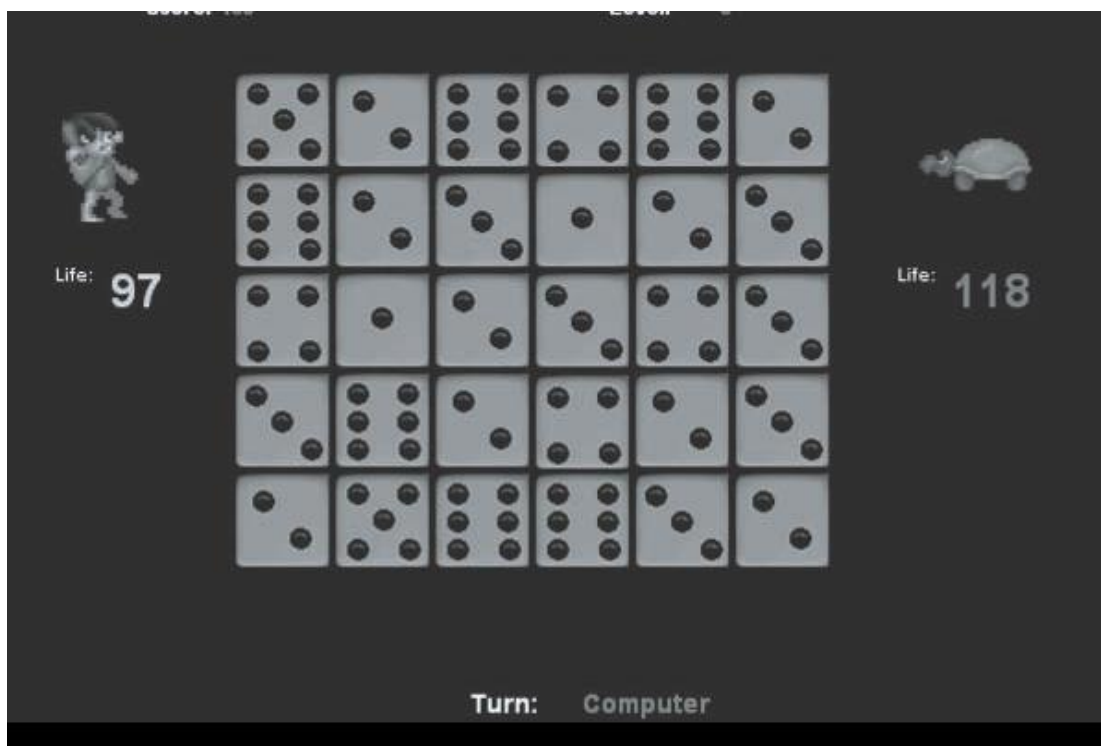


图 9-7 计算机选择移动。

## 等级结束或 game over

游戏状态是 `GameStates.STATE_CHECK_FOR_END` 时会调用 `checkForEndLevel` 和



checkForEndGame方法。

当计算机的生命值(playerLife)小于等于0时当前等级结束。玩家生命值(playerLife)小于等于0时游戏结束。

```
private function checkForEndLevel():Boolean { //changed
var retval:Boolean = false;
if (computerLife <= 0) {
dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND, Main.SOUND_WIN, □
false, 1, 0, 1));
retval = true;
}
return retval;
}

private function checkForEndGame():Boolean { //new
var retval:Boolean = false;
if (playerLife <= 0) {
dispatchEvent( new CustomEventSound(CustomEventSound.PLAY_SOUND, Main.SOUND_LOSE, □
false, 1, 0, 1));
retval = true;
}
return retval;
}
```

## 阅读骰子大战剩余的代码

在这一章，我们只是替换了Color Drop中一些较大修改的代码来建立骰子大战。然而，还是有许多是必须修改的不只是我们简单的用die替换block来符合这个游戏。另一个好的策略是我们从这个环境中抽象出这些方法并创建一个drop-style的游戏类，这个类能被更多的这种类型的游戏继承和复用。

无论如何，剩余的代码与Color Drop中是没有什么区别的 除了现有的变量名和方法名外：

```
private function checkForFallingDice():Boolean {
var falling:Boolean = false;
for (var r:int = 0; r < board.length; r++) {
for (var c:int = 0; c < board[r].length; c++) {
tempDie = board[r][c];
if (tempDie != null) { //Changed Die Battle
```



```
if (tempDie.isFalling) {
    falling = true;
}
}
}
return falling;
}

private function checkForFadingDice():Boolean {
    var fading:Boolean = false;
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            tempDie = board[r][c];
            if (tempDie != null) {
                if (tempDie.isFading) {
                    fading = true;
                }
            }
        }
    }
    return fading;
}

private function replaceDice():void {
    for (var r:int = 0; r < board.length; r++) {
        for (var c:int = 0; c < board[r].length; c++) {
            if (board[r][c] == null) {
                board[r][c] = addDie(r, c);
            }
        }
    }
}

private function addDie(row:Number, col:Number):Die {
    var randomColor:Number = Math.floor(Math.random()*currentLevel.allowedColors.length);
    var dieColor:Number = currentLevel.allowedColors[randomColor];
    var dieValue:Number = Math.floor(Math.random() * 6)+1;
    tempDie = new Die(dieValue, dieColor, tileSheet, row, col, □
        (row*DIE_HEIGHT)+Y_PAD+(row*ROW_SPACING), (Math.random()*10)+10 );
    tempDie.x=(col*DIE_WIDTH)+X_PAD+(col*COL_SPACING);
    tempDie.y= 0 - DIE_HEIGHT;
    tempDie.addEventListener(CustomEventClickDie.EVENT_CLICK_DIE, dieClickListener, □
        false, 0, true);
    this.addChild(tempDie);
    return tempDie;
}
```



```
}  
private function update():void {  
    for (var r:int = 0; r < board.length; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempDie = board[r][c];  
            if (tempDie != null) {  
                if (tempDie.isFalling) {  
                    tempDie.update();  
                }  
            }  
        }  
    }  
}  
  
private function render():void {  
    for (var r:int = 0; r < board.length; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempDie = board[r][c];  
            if (tempDie != null) {  
                if (tempDie.isFalling || tempDie.isFading) {  
                    tempDie.render();  
                }  
            }  
        }  
    }  
}  
  
private function fadeDice():void {  
    var boardLength:int = board.length;  
    for (var r:int = 0; r < boardLength; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempDie = board[r][c];  
            if (tempDie != null) { //Changed Die Battle  
                tempDie.startFade((Math.random()*9)+1);  
            }  
        }  
    }  
}  
  
private function makeDiceFall():void {  
    var boardLength:int = board.length;  
    for (var r:int = 0; r < boardLength; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempDie = board[r][c];  
            if (tempDie != null) { //Changed Die Battle  
                tempDie.startFalling(gameHeight + DIE_HEIGHT, (Math.random()*15)+10);  
            }  
        }  
    }  
}
```



```
}  
}  
}  
}  
  
private function removeDie(rd:Die):void {  
    var boardLength:int = board.length;  
    for (var r:int = 0; r < boardLength; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempDie = board[r][c];  
            if (tempDie == rd) {  
                tempDie.removeEventListener(CustomEventClickDie.EVENT_CLICK_DIE, □  
                    dieClickListener);  
                tempDie.dispose();  
                removeChild(tempDie);  
                board[r][c]= null;  
            }  
        }  
    }  
}  
  
private function cleanUpLevel():void {  
    var boardLength:int = board.length;  
    for (var r:int = 0; r < boardLength; r++) {  
        for (var c:int = 0; c < board[r].length; c++) {  
            tempDie = board[r][c];  
            if (tempDie != null) {  
                removeDie(tempDie);  
            }  
        }  
    }  
}  
  
public function moveDiceDown():void {  
    var collength:int = DIE_COLS-1;  
    for (var c:int = collength; c >= 0; c--) {  
        var rowlength:int = DIE_ROWS-1;  
        var missing:Number=0;  
        for (var r:int = rowlength; r >= 0; r--) {  
            tempDie=board[r][c]  
            if (tempDie != null) {  
                //find number of null spots in this column  
                missing=0;  
                if (r<DIE_ROWS) {  
                    for (var m:int = r + 1;m < DIE_ROWS;m++) {  
                        if (board[m][c]==null) {
```



```
missing++;
}
}
}

if (missing > 0) {
    tempDie.row = r+missing;
    tempDie.col = c;
    board[r+missing][c] = tempDie;
    board[r][c] = null;
    tempDie.startFalling(tempDie.y+(missing*DIE_HEIGHT)+□
(missing*ROW_SPACING), 10);
}
}
}
}
}

public function removeClickedDice():void {
    removeClickedDiceFromScreen();
    moveDiceDown();
    clickedDiceArray = new Array();
}

public function removeClickedDiceFromScreen():void {
    var dieLength:int = clickedDiceArray.length-1;
    while(clickedDiceArray.length > 0) {
        tempDie = clickedDiceArray.pop();
        removeDie(tempDie);
    }
}

private function endGame():void {
    cleanUpLevel();
    dispatchEvent(new Event(GAME_OVER));
}

private function endLevel():void {
    cleanUpLevel();
    dispatchEvent(new Event(NEW_LEVEL));
}
```

你应该在DiceBattle.as的最后括弧内完成类的定义。

```
}
}
```



## 测试

如果你测试骰子大战，你会注意到这个游戏开始非常容易，但是难度提升的太快。你怎样来改变这种情况？

你怎样来创建这个难度等级，使难度提升比较平滑？

有许多方式将这种类型的游戏转变成其他类型的游戏。Color Drop和骰子大战使用了非常简单的游戏引擎，这种引擎能被许多方法进一步扩展。你应该知道些什么当你需要选择多个blocks或者dice一次，在一个卡片游戏里？要是骰子和方块从屏幕不同的方向飞入你又会怎么样？做为选择，你应该重复使用这个游戏中的什么东西在一个角色扮演游戏 Poker Dice puzzle中？

## 总结

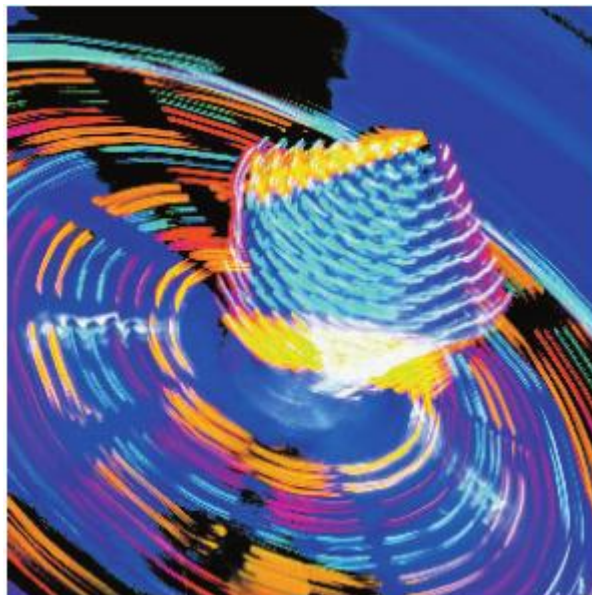
首先，我们讨论了有关游戏法规和知识产权，一个很重要的东西 但是许多将要踏入游戏开发的开发者常常误解。接着我们重复Color Drop改编成一个叫骰子大战的完全不同的游戏。你使用骰子与计算机AI对战。

我们使用的为计算机移动使用的minimax-style AI对于整个计算机AI来说只是冰山一角。如果你想更多的了解有关游戏AI的东西，以下有一些网站可能是对你有用的：

AI 新手站：<http://ai-depot.com/>

AI 杂志：<http://www.aaai.org/ojs/index.php/aimagazine/index>

我们希望最近的 2 章使用对制作这种类型的游戏已经产生兴趣。现在，我们将来制作一个些真正让人兴奋的游戏。



## 第十章

# 滚屏游戏世界

---

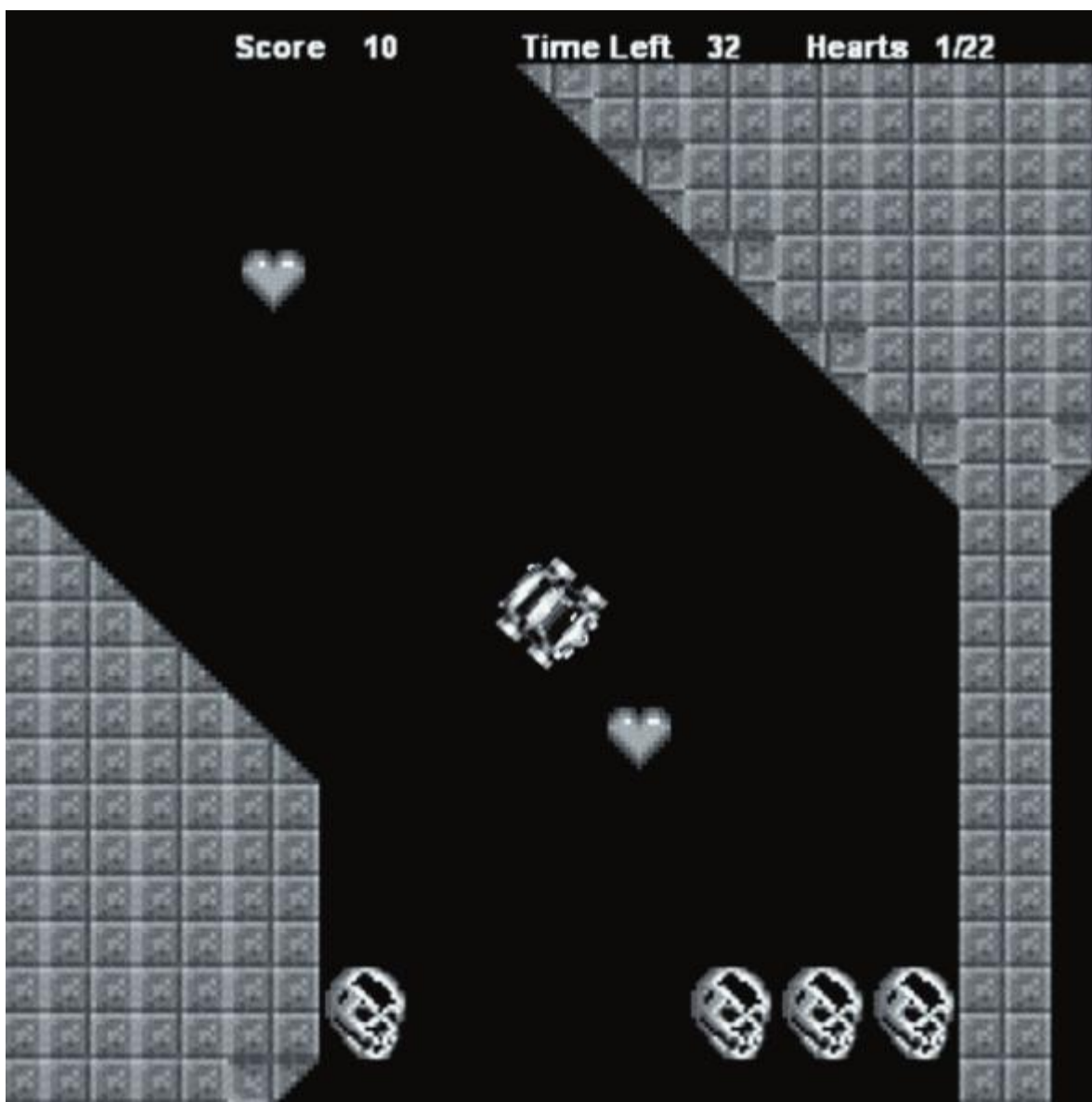
在这章我们开始先介绍区块单屏游戏。理论部分已经在 6, 7 章中有所涉及, 这章主要是把它们变为为滚动的区块世界。我们给这个游起了个名字叫做 Drive She Said。Drive She Said 包含了一个自上而下的场景和一个具有物理特性的小汽车, 你将要操纵这个汽车在滚动的世界行驶。这个汽车是扩展自第七章的 BlitSprite 类, 场景则是一个个显示对象数据组成的区块蒙板, 和第六张谈论的内容比较相似

## 开始设计 Drive She Said

- 让我们先看一下游戏截图(看 10-1), 游戏设计大纲, 和游戏技术规范文档。
- 游戏设计规范如下:



- **游戏名称：**Drive She Said
- **游戏类型：**
- **游戏灵感：**我们玩的第一款竞速类游戏是 Mattel Intellivision 的赛车游戏，这是一个很棒的游戏。
  - 我们特别是 jeff 对于如何制作非常感兴趣。
- **游戏目的：**你会让你的朋友门处于疯狂状态。在时间结束前收集所有的红心。如果你在时间结束前收集完并完成所有的等级，你的聚会会很完美。



[图 10-1]



**如何游戏：**游戏者驾驶着一辆有物理属性的车在滚动的世界中驰骋。拾取红心获得积分结束关卡。头骨会让你的车停下来，钟表会给你增加宝贵的几秒钟。可驾驶的区域会大于一个视屏。游戏者不得不在道路上绕圈直到从起点到终点收集足够多的红心完成关卡

**关卡设计：**不同的关卡需要存储到 XML 又关卡设计者预先设计好

完成过关所需红心的百分比。每个关卡的结束时间，收集一颗红心所得分数，收集时钟增加游戏时间，碰到头骨调整减少速度，碰到墙调整减少速度。

基本技术信息如下：

- 区块大小：32\*32 像素
- 场景大小：50\*50 区块
- 摄像机大小：12\*12 区块
- 滚动区域：
- 用户控制：方向键
- 关卡数据形式：xml
- 区块数据形式：xml

## 这章游戏开发设计的概念

这章在制作 Drive She Said 游戏时涉及的概念

- 扩展 BlitSprite 类实现小汽车的移动。
- 把声音放到类库里面（仅 flex）
- 在滚动的世界中平滑的移动玩家
- 区块在逻辑摄像机里面移动
- 小汽车的基础物理移动

## 定义新类

我们要添加一些新的玩家，帮助和工具类

- ◆ LookAheadPoint:用来检查小汽车和区块之间的碰撞检查
- ◆ Camera2D:
- ◆ CarBlitSprite:扩展自第七章的 BlitSprite 实现小汽车的移动
- ◆ BasicFrameTimer:一个非常简单的时钟计时器。



◆ CustomEventHeartsNeeded:

## 总结一下游戏框架

在 Drive She Said 游戏中我们需要制作以下几个场景

- **标题界面:** 包含一个 play 按钮和显示文字 Drive She Said
- **介绍界面:** 包含一个开始按钮和一个文字 “在时间结束之前获得所有的红心”
- **游戏结束界面:** 包含一个时间最多文本和重新开始按钮

我们需要添加以下计数板元素

- **总分显示**

心的显示，显示收集的心、需要的心

时间显示，显示剩余多少秒

## 修改类库

我们要添加声音到类库消除 flash 的 swf 资源

## 添加自定义关卡类 LevelInScreen

我们要在 Main 中添加代码让 LevelInScreen 动态。这个将允许我们添加下一个文本域告诉玩家这个关卡需要收集的红心。

## 在两个关卡之间变化

Main.as 可以允许修改 Game.as 类实现界面 LevelInScreen 的渐隐

## 理解在 tile 里的自由移动

有许多种方法可以让游戏角色在界面中移动，在第七章，我们创建了一些坦克，游戏中它们平滑的在区格中移动，基于上述我们限制了坦克的移动在两个区格的中心线之间平滑移动。这章里面我们要使用不同的区格移动被称做自由区格移动



## 定义自由区格移动的工作

在自由区格移动中，游戏世界的角色可以移动和并且以像素点大小进行碰撞检测，需要的代码和 6，7 章的基本上不一样。

我们要放弃所有在区格中的检查取而代之的是使用系统观测点而不是下一个区块能否让玩家进入。

我们的小汽车可以在俯视图 360 度旋转和移动。我们通常称这种移动为 “n-way movement”，由于玩家可以任意方向移动而不是像第六章和第七章那样仅仅在四个方向上移动。

区块滚动是游戏在超出界面后，重新建立一部分游戏世界，游戏世界的建立依靠的是 2d 区格数组的值

使用艺术手法实现舞台场景的滚动

传统的滚动方式（至少实在 flash 中是这样），游戏世界会设置在一个 MovieClip 代表整个世界，这个 MovieClip 需要在主时间轴上根据上下左右的滚动情况来移动。因此，主时间轴上的显示窗体必须要比游戏世界小。

**备注：**最新的 flash player 版本已经把 art-based 滚动。。。他现在已经可以在舞台中更新成百上千的 MovieClips 和其它可见的被忽略的可见空间。

## GAS (GotoAndStop) tiles

一种解决方案是使用时间轴。最好的表现方式是使用一个 2d 区格数组。把不同的区格（路，建筑）等分别放到 tile 影片的帧上面。使用 “Go to and stop” 方法跳到相应的帧上，改变显示。当不同的区格放到界面上，一个区格的世界就被创建了。“Go to and stop” (GAS)涉及 flash 时间轴控制命令 MovieClip.gotoAndStop。

我们可以认为 GAS 方法是很早的。每个 tile 被放到影片的不同帧里面。最基本的创建一个

并且该方法有很多种不同的实现形式，一些人把所有的 tiles 放到时间轴的不同帧上，而另一些人仅仅把需要的放到界面上，并且只是改变需要变化的。当用 flash 开发动作快，速度高的游戏是。这两种方法都显示了不足之处。flash 引擎不能解决在很高的帧频的情况下让使用大量的 MovieClips 来创建一个世界。

## 使用 tile-based blit scrolling

还有一小部分的人使用 blit canvas 来布置界面。这章里面我们讨论使用 tile-based blit scrolling。在下一章我们将要探索 screenbased blit scrolling。Tile-based blit scrolling 的概念与 GAS 非常相似，但是可以支持搞帧频。我们将要使用 2D 数组描述这个世界。我们不会一进入游戏就显示所有的东西。我们只要显示当前摄像机需要的 tiles。滚动 buffer 蒙板按照精确的 32\*32 的 tile 来输出结果



## 场景世界

场景世界是一个 2d 数组 ( 50\*50 ) 组成的。每个 tile 赋予一个值像墙，路，心，头盖骨 ( 与第六章和第七章的坦克非常像 )

## 摄像机

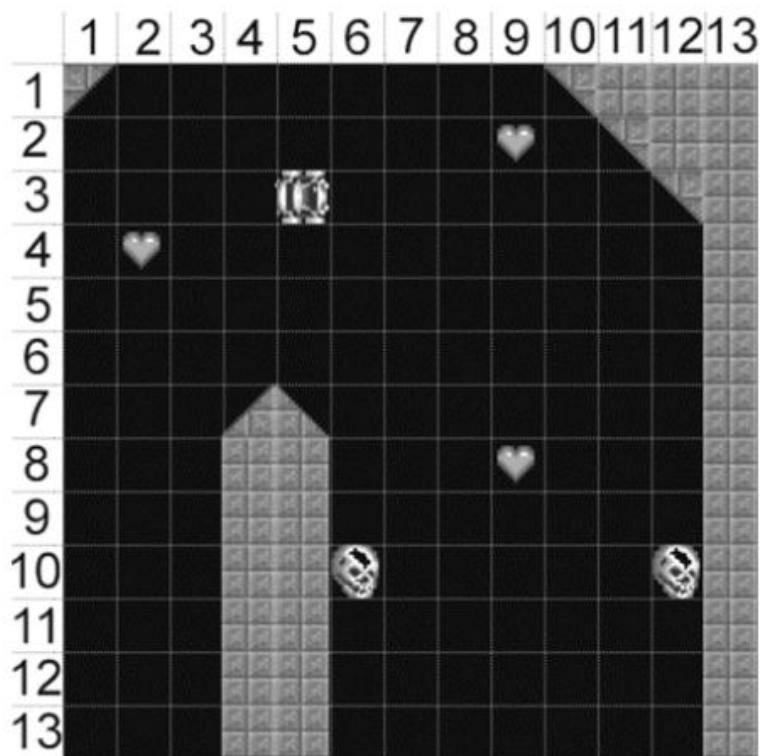
摄像机是用来设定变量

the buffer

buffer 是一个 bitmapdata canvas 由 13 个 tiles

## 输出界面

输出的界面是 12\*12 ( 32\*32 tiles = 384\*384 面积 )。起点开始于左上角。输出界面从 buffer 中 copy 数据。13 行 13 列我们从右向下开始我们的数据 copy。在每个方向上仅仅显示 384\*384 像素。



图像 10-2 缓存图像数据

图像 10-2 显示 canvas 缓存的内容输出到屏幕的例子。随着小汽车从左向右移动。我们必须滑动左上角的。这列的摄像但是个

起点。正如你所见到个缓存包含 13 行 13 显示数据例如实际的机没有很准确的界限，起点在 10x12y 在第一 tile。

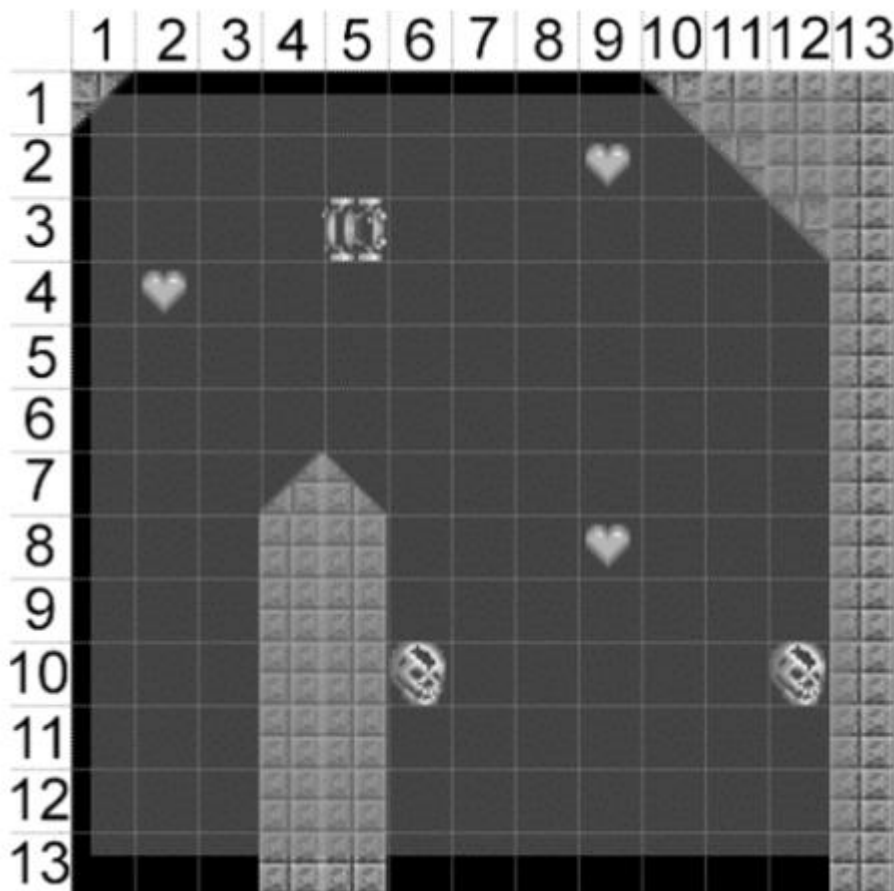


图 10-3 显示了那些东西需要显示到界面上

图 10-3 数据图像矩形缓冲区

图 10-3 中的阴影部分是备份数据输出到界面的实际区域。它表现了实际 x 和 y 坐标在摄像机中.这也就是为什么 buffer 需要 13 行\*13 列我们用这种方法实现场景平滑过渡。

## 创建游戏世界

我们再一次的使用 Ari Feldman' s Spritelib GPL 软件。我们在可变的 sheets 里面挑出变化少的组成一个 sheet。我们根据需要修改他们。你也可能注意到了我们用一些区格做带转角的墙。



## 创建区格表

图 10-4 显示的是 32\*32 大小的 tile 表。我们将要在这个游戏中使用它。



前三个区格表示玩家的  
小汽车轮胎前进方向动  
画。我们要让小车的移动

动画随小车的速度变化而变化。随后我们将讨论给小车加上物理属性。

紧跟着小车的是红心，时钟，骷髅头，然后是 15 个不同形状的墙。所有的黑色 tile 表示可以行走的道路。最后两个 tile 表示不同方向的结束线。

## 关于墙体的碰撞检测

15 个灰色方格和一个透明的背景将要组成我们游戏世界的墙体。所有黑色的区域显示了我们可以通过。黑颜色非常重要，不仅仅是因为它和路的颜色相匹配。这个颜色可以用来描绘和墙体的区别。如果我们简单的使用区块间的碰撞检测，那么小车只要碰到墙体 tile 上任何一个点都会停下来。这种情况对于 tile 完全是墙体的比较好。其他的部分墙体的表示就不好了。

所以我们使用黑颜色包围在 tile 周围作为碰撞缓冲区。如果小车碰撞到墙上，但是这个像素点的颜色是黑色。这个碰撞点就被舍弃。颜色缓冲区可以选择性的设置关卡。当然还有一些其它的原因为什么代码不能提供判断更多的颜色。因为颜色的值是 32 位，你可以使用透明来创建无缝世界。甚至黑颜色。我们创建一个漂亮的无缝环境。你不用限制使用简单的颜色来检测碰撞。例如，我们可以画一个不透明的像素墙砖颜色。你可以在关卡的 xml 里面改变颜色或者是改变 game.as。一个是通过墙的边缘检测碰撞是否。

注意，游戏背景将单独分在一个层里面。通过这个，我们根据需要重绘背景。这也、



像第六七章的游戏关卡，我们使用 Mappy 来创建我们的关卡。看一下第六章使用 Mappy 设置的。你不一定使用 Mappy 或者其它工具。

不像第六七章，我们不会创建两个层，在游戏里面仅有一个 sprite,那就是玩家的汽车。每个事情，特别是能够动的物体都是游戏背景的一部分。这样我们就可以很简单的滚动屏幕来，而不用保证所有游戏对象的相对位置。

在我们的关卡设计规则里面，每个关起包含如下的信息

50 行 50 列的  $32 \times 32$  的 tiles。

每个关卡的墙体都报场景包了起来（跟 Tanks 那章不一样）

仅有一个

至少有一个红心

至少有一个黄色的终点线

确保所有的黑色 tiles 有 22 个，而不是 0 个 tile 从 Mappy。

像 No Tanks 的关卡数据，我们的关卡存储在 2d 数组里面，使用自定义菜单的“导出 ActionScript”选项。每个 tile 的数据向下面摘录的这样。6 是 tile 的数字。tile 行必须在场景的最上面所以 rowid 是 6。

[illegible]

我们创建的场景要显示的足够大。

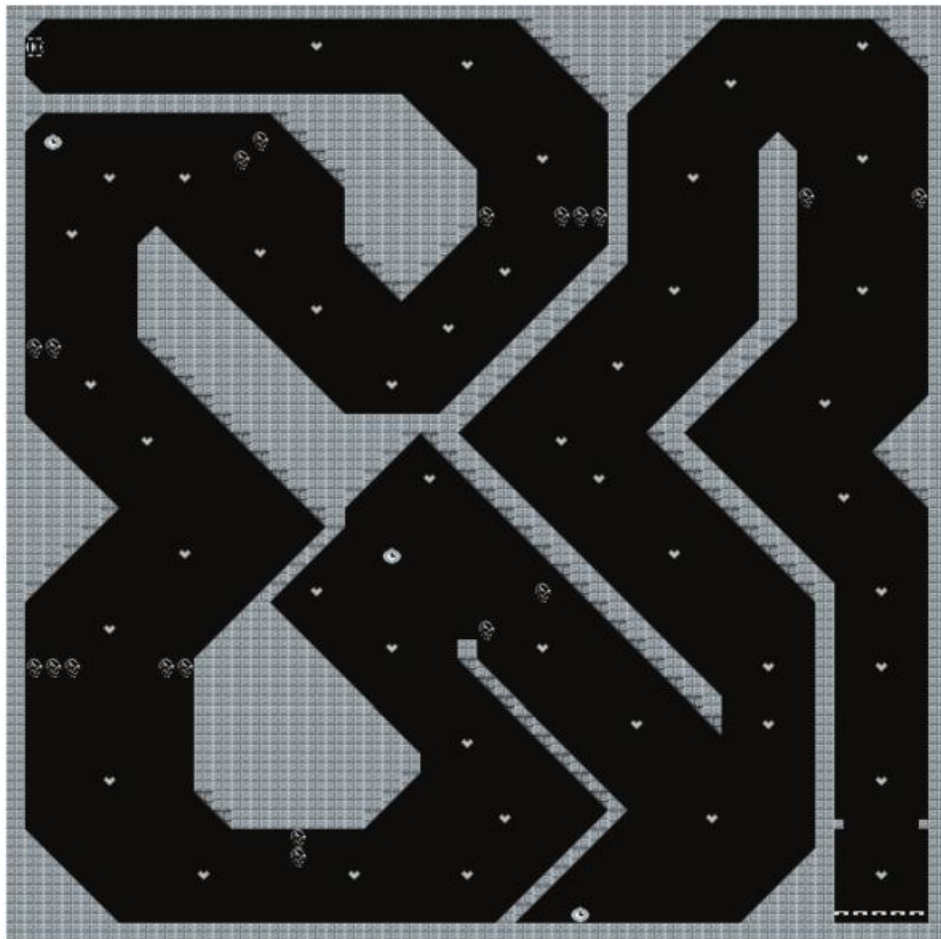


图 10-5 是多个图像拼接在一起组成的第一关。你可以按照第一关模拟创建你自己的游戏数据



**注意：**Drive She Said 引擎，可以被用来制作循环。

## 给小汽车添加物理方式

我们想做一个小汽车让游戏者可以觉得它开始移动和停止的过程。我们不会模拟过多的真实汽车的运动。我们只要给我们的游戏增加点真实感就足够了。我们曾在第六七章中探讨过让坦克在迷宫中穿行。我们不会过多地讨论细节。我们只是大概的涉及一下。

## 向前向后运动

当我们的汽车停止的时候，它的速度为 0。当方向键被按下后，我们要加速增加速度。这个速度应该有一个最大值（如 8 像素）当达到最大速度的时候，小车就不能在加速了。当松开方向键后，速度则会慢慢的减为 0。当按下后推键时，玩家会逐渐的停下来并且开始朝反方向运动。有一个最大的反向加速度。我们的汽车不能够反转或者是拐弯除非他有向前和反向速度。

## 朝一个方向移动

我们的速度值仅仅是个汽车当前速度的数量值。实际上，速度是个矢量值包括数值和方向  $x, y$ 。

小汽车的运动方向需要考虑矢量值。一个矢量值由大小和方向组成。我们有这样的一个变量叫做 velocity，但是它的实际值是我们需要值的一半。我们的矢量运动值存储包括数值和表示  $xy$  方向的角度。数值表示大小，角度表示方向。那么在水平方向我们有一个值是 -1。它表示向左的 1 像素。我们可能有一个在垂直方向是 3（没有符号表示的是正值）它表示向上 3 个像素。

当用方向键控制汽车转弯或着旋转到一个新方向的时候。我们需要计算新方向的矢量值。要想计算出新方向的矢量值我们需要知道小汽车将要移动的角度。

我们使用两个值来存储汽车的移动方向。一个存储的是  $x$  方向，一个是存储的是  $y$  方向。这两个值有时候表示为  $dx$  和  $dy$ 。它们表示在每个方向上改变。

我们现在做一个例子，让我们的汽车旋转 30 度，现在我们需要计算  $dx$  和  $dy$  在下一帧的值。

让我们看一下样例代码通过调试输出来阐述上述观点



```
var velocity:Number = 2;
var rotation:Number = 30;
var carRadians:Number = (rotation / 360) * (2.0 * Math.PI);
trace("rotation=" + rotation);
trace("carRadians=" + carRadians);
trace("Math.cos(carRadians)=" + Math.cos(carRadians));
trace("Math.sin(carRadians)=" + Math.sin(carRadians));
var dx:Number=Math.cos(carRadians)*velocity;
var dy:Number= Math.sin(carRadians) * velocity;
trace("dx=" + dx);
trace("dy=" + dy);
```

这里是输出信息

```
rotation=30
carRadians=0.5235987755982988
Math.cos(carRadians)=0.8660254037844387
Math.sin(carRadians)=0.49999999999999994
dx=1.7320508075688774
dy=0.9999999999999999
```

在输出窗口有一些关键的事情

首先，flash 使用度数来表示显示对象的角度，这样我们必须把这个值转变成为弧度来用于计算

$\text{Radians} = (\text{rotation in degrees} / 360) * (2 * \pi)$

那么我们的例子

```
Radians = 30/360 * (2 * pi)
Radians = .083 * (6.28)
Radians = .0523
```

下一步，我们查找 x 轴上的速度。如果我们记得基本三角学（如果你有），在 X 轴和余弦的角度密切相关。角度的余弦值

dx 值是余弦乘以速度。当我们的车速度为 2（在这个例子里面），我们这样计算

```
dx=cos(.523) * 2
dx=.866*2
dx=1.73;
```

现在我们看一下 dy 的速度，这个轴与正弦值密切相关。正选角度是以像素为级别让我们的小车在 y 轴上移动

```
dy=sin(.523)*2
dy=.499 * 2
```



dy=0.999

以 30 度的角度移动我们的小车，我们需要移动 1.73 像素沿着 x 轴和 0.999 像素沿着 y 轴

## 准备创建我们的 Drive She Said 游戏

我们的游戏使用框架类（做了稍许改动）我们要自定义一些新类，这些在这章的先前已经提到了。

不管你是使用 flex，flash develop，flash builder 或者是 flash ide 都没有问题，你可以新建一个工程来保存该游戏的所有代码。

就像这本书所有的游戏，Drive She Said 使用了第二章创建的架构类库。让我们开始使用 flash ide 或者是 flash develop 来创建我们需要的包（也可以使用 flex sdk）

## 在 flash ide 中创建游戏项目

现在，这里有几步使用 flash ide 创建项目

- 1.开始选择 flash 版本，我们使用 cs3，当然在 cs4，cs5 中也一样。
- 2.创建一个 a.flah 文件在/source/projects/driveshesaid/flashIDE/文件夹里面
- 3.在/source/projects/driveshesaid/flashIDE 文件夹，创建 com/efg/games/driveshesaid/包结构
- 4.设置 flash 影片的帧率 30fps。设置宽度为 384 高度为 404.
- 5.设置文档类 com.efg.games.driveshesaid.Main
- 6.由于我们还没有创建 Main.as 类，所以我们会看到一些警告。我们将在本章的后面介绍
- 7.现在我们需要在 fla 文件里面添加框架类库路径。在发布设置选择 flash-》actionscript3 设置。
- 8.单击浏览路径按钮，查找/source 包路径
- 9.选择类库所在的路径，单击选择按钮，现在 com.efg.framework 包可以在我们的游戏中使用了

在 flash develop 中创建游戏工程

下面是用 flash develop 创建工程的步骤

- 1.在/source/projects/driveshesaid/文件夹里面创 flexSDK 文件夹。

2.开始用 flash develop 创建一个新的项目；选择 flex3 项目，写上名字 driveshesaid。本地路径 /source/projects/driveshesaid/flexSDK，包名 com.efg.games.driveshesaid。不要让 flash develop 自动建立项目。确定创建项目目录是没有勾选的。单击 ok 按钮创建工程



3.现在我们需要给项目添加框架类库路径。选择项目-》属性-》类路径菜单项

4.添加单击类路径按钮。找到早先我们创建的源文件，选择类所在的文件夹。

5.单击 ok 按钮然后单击提交。

6.下一步，改变输出帧率大小。再一次选择项目-》属性-》类路径菜单项，设置帧率为 30，宽度为 384，高度为 404

你现在有一个基本结构开始创建项目。我们现在讨论一下框架类的结构，然后建立可重复使用的代码。

在 flex builder， flash builder 或者是其它 ide，请按照文档提供的方式来建立新的项目设置默认的编译类。

在 flash develop/flash ide 工作流中。一个普通方法

这里是 flash ide 的结构

```
/source/projects/driveshesaid/flashIDE/com/efg/games/driveshesaid/
```

这里是 flex sdk ( 使用 flash develop )

```
/source/projects/driveshesaid/flexSDK/
```

```
assets/
```

```
bin/
```

```
obj/
```

```
lib/
```

```
src/com/efg/games/driveshesaid/
```

## 编写 Drive She Said 的 Main.as 类

我们将要在 Main.as 里面做重要的更改。它仍旧是 GameFrameWork.as 的子类，但是我们添加一些新的功能行缓动和移动游戏界面。

这里是 flash ide 的文件名

```
/source/projects/driveshesaid/flashIDE/com/efg/games/driveshesaid/Main.as
```

这里是 flex sdk 的文件名



---

/source/projects/driveshesaid/flexSDK/src/com/efg/games/driveshesaid/Main.as

下面这个是 Drive She Said 的 Main.as 类

```
package com.efg.games.driveshesaid
{
import flash.text.TextFormat;
import flash.text.TextField; //new
import flash.text.TextFormatAlign;
import flash.geom.Point;
import flash.events.Event;
import com.efg.framework.FrameWorkStates;
import com.efg.framework.GameFrameWork;
import com.efg.framework.BasicScreen;
import com.efg.framework.ScoreBoard;
import com.efg.framework.SideBySideScoreElement;
import com.efg.framework.SoundManager;
public class Main extends GameFrameWork {
//custom score board elements
public static const SCORE_BOARD_SCORE:String = "score";
public static const SCORE_BOARD_TIME_LEFT:String = "timeleft";
public static const SCORE_BOARD_HEARTS:String = "hearts";
//custom sounds
public static var SOUND_TITLE_MUSIC:String = "titlemusic";
public static var SOUND_CAR:String = "car"
public static var SOUND_CLOCK_PICKUP:String = "clockpickup";
public static var SOUND_HEART_PICKUP:String = "heartpickup";
public static var SOUND_GAME_LOST:String = "gamelost";
public static var SOUND_LEVEL_COMPLETE:String = "levelcomplete";
public static var SOUND_SKULL_HIT:String = "skullhit";
public static var SOUND_PLAYER_START:String = "playerstart";
public static var SOUND_HIT_WALL:String = "hitwall";
//level in screen additions
private var heartsToCollect:TextField = new TextField();
public function Main() {
init();
}
override public function init():void {
game = new DriveSheSaid();
game.y = 20;
game.x = 404; //added
setApplicationBackGround(384, 404, false, 0x000000);
game.addEventListener(CustomEventHeartsNeeded.HEARTS_NEEDED, heartsNeededListener,0, false, 0, true);
```



```
//add score board to the screen as the seconf layer
scoreBoard = new ScoreBoard();
addChild(scoreBoard);
scoreBoardTextFormat = new TextFormat("_sans", "11", "0xffffffff", "true");
scoreBoard.createTextElement(SCORE_BOARD_SCORE, new SideBySideScoreElement(
(80, 5, 20,"Score", scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_TIME_LEFT, new SideBySideScoreElement(
(180, 5, 20, "Time Left", scoreBoardTextFormat, 45, "0", scoreBoardTextFormat));
new Point(150, 250), 2
100, 20,screenButtonFormat, 0x000000, 0xff0000,2);
instructionsScreen.createDisplayText("Drive over all harts\nbefore 2
timer\nruns out.",200,new Point(100,150),screenTextFormat);
gameOverScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER,2
640,500,false,0x0000dd);
gameOverScreen.createOkButton("Restart", new Point(150, 250), 100, 20,2
screenButtonFormat, 0x000000, 0xff0000,2);
gameOverScreen.createDisplayText("Time up\nGame Over",100,new Point(150,150),2
screenTextFormat);
levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN, 2
384, 404, true, 0xbbff00ff);
levelInText = "Level ";
levelInScreen.createDisplayText(levelInText,100,new Point(150,150),2
screenTextFormat);
heartsToCollect.defaultTextFormat = screenTextFormat;
heartsToCollect.width = 300;
heartsToCollect.x = 50;
heartsToCollect.y = 200;
levelInScreen.addChild(heartsToCollect);
//set initial game state
switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
//sounds
//*** Flex SDK
soundManager.addSound(SOUND_TITLE_MUSIC,new Library.SoundTitleMusic);
soundManager.addSound(SOUND_CAR, new Library.SoundCar);
soundManager.addSound(SOUND_CLOCK_PICKUP,new Library.SoundClockPickup);
soundManager.addSound(SOUND_HEART_PICKUP,new Library.SoundHeartPickup);
soundManager.addSound(SOUND_GAME_LOST,new Library.SoundGameLost);
soundManager.addSound(SOUND_LEVEL_COMPLETE,new Library.SoundLevelComplete);
soundManager.addSound(SOUND_SKULL_HIT,new Library.SoundSkullHit);
soundManager.addSound(SOUND_PLAYER_START,new Library.SoundPlayerStart);
soundManager.addSound(SOUND_HIT_WALL,new Library.SoundHitWall);
//*** Flash IDE SDK
//soundManager.addSound(SOUND_TITLE_MUSIC,new SoundTitleMusic);
```



---

```
//soundManager.addSound(SOUND_CAR, new SoundCar);
//soundManager.addSound(SOUND_CLOCK_PICKUP,new SoundClockPickup);
//soundManager.addSound(SOUND_HEART_PICKUP,new SoundHeartPickup);
//soundManager.addSound(SOUND_GAME_LOST,new SoundGameLost);
//soundManager.addSound(SOUND_LEVEL_COMPLETE,new SoundLevelComplete);
//soundManager.addSound(SOUND_SKULL_HIT,new SoundSkullHit);
//soundManager.addSound(SOUND_PLAYER_START,new SoundPlayerStart);
//soundManager.addSound(SOUND_HIT_WALL,new SoundHitWall);
//create timer and run it one time
frameRate = 40;
startTimer();
}
override public function systemTitle():void {
soundManager.playSound(SOUND_TITLE_MUSIC, false,999, 20, 1);
super.systemTitle();
}
override public function systemNewGame():void {
soundManager.stopSound(SOUND_TITLE_MUSIC);
super.systemNewGame();
}
override public function systemLevelIn():void {
levelInScreen.alpha = 1
super.systemLevelIn();
}
override public function systemWait():void {
if (lastSystemState == FrameWorkStates.STATE_SYSTEM_LEVEL_IN) {
game.x -= 2;
if (game.x < 100) {
levelInScreen.alpha -= .01;
if (levelInScreen.alpha < 0 ) {
levelInScreen.alpha = 0;
}
}
if (game.x <= 0) {
game.x = 0;
soundManager.playSound(SOUND_PLAYER_START, false,1,20, 1);
dispatchEvent(new Event(EVENT_WAIT_COMPLETE));
}
}
private function heartsNeededListener(e:CustomEventHeartsNeeded):void {
heartsToCollect.text = "Collect " + e.heartsNeeded + " Hearts";
heartsToCollect.width = 300;
heartsToCo, tsToCollect.y = 200;
```



```
}  
}  
}
```

我们要给 Main 类做一个变动开关，让它可以改变与框架的交互方式，我们通过浏览整本书，已经把游戏同类型的整体话

## 添加积分牌元素

到目前为止所有的游戏，我们添加积分牌在界面的上方；有三个 SCORE\_BOARD\_SCORE 将要显示收集红心获得的值。SCORE\_BOARD\_TIME\_LEFT 将显示每关剩下的红心的数。SCORE\_BOARD\_HEARTS 显示到目前为止收集的红心

个数。

```
public static const SCORE_BOARD_SCORE:String = "score";  
public static const SCORE_BOARD_TIME_LEFT:String = "timeleft";  
public static const SCORE_BOARD_HEARTS:String = "hearts";
```

## 修改界面

界面会根据我们的需要显示不同的信息。这些都是非常简单非常直接的和以往的游戏一样。

## 修改声音

我们在 SoundManager 里面添加 9 种不同的声音，和前几章不一样的是，我们使用 flex sdk 的绑定技术把所有的声音打包成 swf，我们将直接使用 MP3 文件。Flash IDE 将在用 flex sdk 编译时将这些资源会被直接绑定到类库中。

```
/** Flex SDK  
soundManager.addSound(SOUND_TITLE_MUSIC,new Library.SoundTitleMusic);  
soundManager.addSound(SOUND_CAR, new Library.SoundCar);  
soundManager.addSound(SOUND_CLOCK_PICKUP,new Library.SoundClockPickup);  
soundManager.addSound(SOUND_HEART_PICKUP,new Library.SoundHeartPickup);  
soundManager.addSound(SOUND_LEVEL_COMPLETE,new Library.SoundLevelComplete);  
soundManager.addSound(SOUND_SKULL_HIT,new Library.SoundSkullHit);  
soundManager.addSound(SOUND_PLAYER_START,new Library.SoundPlayerStart);  
soundManager.addSound(SOUND_HIT_WALL,new Library.SoundHitWall);  
soundManager.addSound(SOUND_GAME_LOST,new Library.SoundGameLost);  
/** Flash IDE SDK  
//soundManager.addSound(SOUND_TITLE_MUSIC,new SoundTitleMusic);
```



```
//soundManager.addSound(SOUND_CAR, new SoundCar);  
//soundManager.addSound(SOUND_CLOCK_PICKUP,new SoundClockPickup);  
//soundManager.addSound(SOUND_HEART_PICKUP,new SoundHeartPickup);  
//soundManager.addSound(SOUND_GAME_LOST,new SoundGameLost);  
//soundManager.addSound(SOUND_LEVEL_COMPLETE,new SoundLevelComplete);  
//soundManager.addSound(SOUND_SKULL_HIT,new SoundSkullHit);  
//soundManager.addSound(SOUND_PLAYER_START,new SoundPlayerStart);  
//soundManager.addSound(SOUND_HIT_WALL,new SoundHitWall);
```

如果你使用的是 flahs ide 需要把 flex sdk 的 addsound 调用方法注释掉，把 flahs ide 的注释去掉

像前几章，我们复写 systemTitle 和 systemNewGame 方法处理标题画面的音乐。

## 过渡到关卡界面

这个版本的 Main 最大的变化是，我们的 game 类实例的起始位置是 404.

```
game.x = 404;
```

我们也要覆盖 systemWait 方法。

在 STATE\_SYSTEM\_LEVEL\_IN 状态，这个新方法会移动游戏界面从 x 轴右上角到 0，游戏玩家开始的地方。当然这个游戏的 levelInScreen 淡出决定与他的透明度值

```
if (lastSystemState == FrameWorkStates.STATE_SYSTEM_LEVEL_IN) {  
    game.x -= 2;  
    if (game.x < 100) {  
        levelInScreen.alpha -= .01;  
        if (levelInScreen.alpha < 0 ) {  
            levelInScreen.alpha = 0;  
        }  
    }  
    if (game.x <= 0) {  
        game.x = 0;  
        soundManager.playSound(SOUND_PLAYER_START, false,1,20, 1);  
        dispatchEvent(new Event(EVENT_WAIT_COMPLETE));  
    }  
}
```

注意，一旦 levelInScreen 到达结束位置，我们就要开始播放声音

我们还要覆盖 systemLevelIn 方法，设置 levelInScreen 的透明属性在回到 1.

```
override public function systemLevelIn():void {  
    levelInScreen.alpha = 1  
    super.systemLevelIn();  
}
```



```
}
```

## 更新 LevelInScreen 的文本信息

我们添加了一个新的消息到 LevelInScreen 上。这个消息显示了红心的数量，需要在这关收集的数量。开始我们确保导入 TextField 类

```
import flash.text.TextField;
```

下一步，我们添加 heartsToCollect 变量

```
private var heartsToCollect:TextField = new TextField();
```

在初始化方法里面我们设置 heartsToCollect 的 defaultTextFormat 属性和界面文字的的属性一样

```
heartsToCollect.defaultTextFormat = screenTextFormat;
```

```
heartsToCollect.width = 300;
```

```
heartsToCollect.x = 50;
```

```
heartsToCollect.y = 200;
```

heartsToCollect 文本域需要添加到 levelInScreen 显示列

```
levelInScreen.addChild(heartsToCollect);
```

我们在初始化方法中为新的自定义事件 CustomEventHeartsNeeded 添加监听事件。当这个事件分发的时候，这关玩家需要收集红心的数量由他携带发送。

```
game.addEventListener(CustomEventHeartsNeeded.HEARTS_NEEDED, heartsNeededListener, 0, false, 0, true);
```

heartsNeededListener 方法需要添加到 Main.as

```
private function heartsNeededListener(e:CustomEventHeartsNeeded):void {
```

```
heartsToCollect.text = "Collect " + e.heartsNeeded + " Hearts";
```

```
}
```

这个方法改变 heartsToCollect.text 的值

## 增加游戏的帧频

当我们需要越来越多的游戏资源时，我们将要使用 Timer 类创建不同的帧频。我们要设置帧频为 40 记得 GameFrameWork 游戏循环 updateAfterEvent 方法。下一章，我们创建一个新的时间，来限制舞台的帧品。

```
frameRate = 40;
```



## 创建 CustomEventHeartsNeeded.as 类

这个类比我们在第二章创建的自定义事件类。这是 Drive She Said 的类包的一部分而不是框架的一部分。

这个文件名和 flash ide 本地：

/source/projects/driveshesaid/flashIDE/com/efg/games/driveshesaid/CustomEventHeartsNeeded.as

这是 flex sdk 文件名和本地（使用 flash develop）

/source/projects/driveshesaid/flexSDK/src/com/efg/games/driveshesaid/CustomEventHeartsNeeded.as

这个类是 Event 的子类,这是它的全代码

```
package com.efg.games.driveshesaid
{
import flash.events.Event;
/**
 * ...
 * @author Jeff Fulton
 */
public class CustomEventHeartsNeeded extends Event{
public static const HEARTS_NEEDED:String = "hearts needed";
public var heartsNeeded:String;
public function CustomEventHeartsNeeded(type:String,heartsNeeded:String,bubbles:Boolean=false,cancelable:Boolean=false) {
super(type, bubbles,cancelable);
this.heartsNeeded = heartsNeeded;
}
public override function clone():Event {
return new CustomEventHeartsNeeded(type,heartsNeeded, bubbles,cancelable)
}
}
}
```

当类的实例被创建以后 heartsNeeded 值被传了进来。监听函数，我们获得 newLevel 方法在 DriveSheSaid 类。

## 创建 Library.as 类

Library.as 类仅仅只能用在 flex 框架下它不支持在 flash ide 项目。运行库的改变之处是，以静态常量



的形式添加声音而不是由 ide 内部导出的 swf。这样我们就可以根据需要进行选择 ide。使用资源运行时绑定的缺点是 flex 框架不能导入 wav 文件。使用 mp3 文件可以减缓限制，但是令人头疼的是如何避免 MP3 开始的空白区域。我们使用偏移值跳过空白区域播放声音。

这里是关于 Mp3 空白区域：Mp3 格式的文件开始有一段 ID3 标签和一些特别的信息。当我们播放文件时，有一段短暂的空白在声音开始播放的时候。当我们不循环播放的时候，这个空白区域几乎不起眼。当你循环播放或者是按次数播放的话，空白区域在结束和再次开始之间就会很明显。幸运的是，as3 的声音处理函数允许我们跳过空白播放区域

一些 mp3 文件编码的时候已经把空白区域在尾部去掉了。不幸的是 as3 没有内建的处理方式。

如果你使用 flash ide，你可以导入声音到库里面，添加一个类名与它相关联。在 flash ide 里面你不需要 library 类。

下面是 flex sdk 的文件结构

/source/projects/driveshesaid/flexSDK/src/com/efg/games/driveshesaid/Library.as

使用下面的代码创建文件

```
package com.efg.games.driveshesaid
{
    public class Library {
        [Embed(source = "../assets/tile_sheet.png")]
        public static const TileSheetPng:Class;
        [Embed(source="../assets/sound_titlemusic.mp3")]
        public static const SoundTitleMusic:Class;
        [Embed(source="../assets/sound_car.mp3")]
        public static const SoundCar:Class;
        [Embed(source="../assets/sound_clockpickup.mp3")]
        public static const SoundClockPickup:Class;
        [Embed(source="../assets/sound_heartpickup.mp3")]
        public static const SoundHeartPickup:Class;
        [Embed(source="../assets/sound_gamelost.mp3")]
        public static const SoundGameLost:Class;
        [Embed(source="../assets/sound_levelcomplete.mp3")]
        public static const SoundLevelComplete:Class;
        [Embed(source="../assets/sound_skullhit.mp3")]
        public static const SoundSkullHit :Class;
        [Embed(source="../assets/sound_playerstart.mp3")]
        public static const SoundPlayerStart:Class;
        [Embed(source="../assets/sound_hitwall.mp3")]
        public static const SoundHitWall:Class;
```



```
}  
}
```

你必须确保你把 tile sheet 和音乐资源放到/assets 文件夹下。并把这个文件夹放到下面的结构里面

```
[source]  
[projects]  
[driveshesaid]  
[flexSDK]  
[assets]  
[bin]  
[obj]  
[lib]  
[src]  
[com]  
[efg]  
[games]  
[driveshesaid]
```

记住，资源文件夹和 src,bin,obj,lib 文件夹在同一个目录下。它不会自动建立；你需要自己创建它。

## 创建新类

我们要创建一些新类，可以在以后的项目中复用。这些类分别是 BasicFrameTimer ,LookAheadPoint , CarBlitSprite , Canvas2D。

## 创建 BasicFrameTimer 类

BasicFrameTimer 被用来创建倒计时。这个类非常简单，主要记录剩下的时间。它没有任何可视化接口，只有逻辑数据。这样的话他需要扩展 EventDispatcher 类。

它不需要使用 getTimer 类，因为混合时间和帧频对于机器慢的玩家不公平。以真实时间倒计时，这个时钟会很准确，但如果游戏速度变慢，玩家可能不能按时到达目标。混合时间和帧频代码可以让游戏按正常的速度运行。

我们使用帧频的方式倒计时。我们游戏的构造方法需要接收一个参数来设置帧频。一分一秒的时间是基于 1000 毫秒。如果帧频快了或者慢了，我们帧频的秒数也会相应的变化。这样可以让玩家在不同的机器上有相同的结果。

这是本地文件路径

/source/classes/com/efg/framework/BasicFrameTimer.as

这是代码



```
framesPerSecond:int;
public function BasicFrameTimer(framesPerSecond:int) {
    this.framesPerSecond = framesPerSecond;
}
public function start():void {
    frameCount = 0;
    started = true;
}
public function stop():void {
    started = false;
}
public function reset():void {
    if (countUp) {
        seconds = max;
    }else {
        seconds = min;
    }
}
public function update():void {
    frameCount++;
    if (started) {
        if (frameCount > framesPerSecond) {
            frameCount=0;
            if (countUp) {
                seconds++;
                if (seconds == max) {
                    stop();
                    dispatchEvent(new Event( TIME_IS_UP));
                }
            }else {
                seconds--;
                if (seconds == min) {
                    stop();
                    dispatchEvent(new Event( TIME_IS_UP));
                }
            }
        }
    }
}
```

这个类接收一个参数来设置每秒帧频，这让用户很容易设置开始和结束的最大值和最小值。它包含一个属性叫做 countUp 开始设置为 false。游戏循环调用的 update 方法必须调用它。在循环更新中，这个



类使用 frameCount 变量在等于 framesPerSecond 是记录值。我们准备一秒更新一次。如果时间变量达到最大值或最小值 TIME\_IS\_SUP 就会抛出事件。Game 监听这个消息之后设置 DriveSheSaid 的 gameOver 变量为 true。

## 创建 LookAheadPoint 类

LookAheadPoint 类是用来检测玩家汽车之间和汽车与墙体之间碰撞检测的。如果小汽车以正方向的速度运动我们在三个方向上使用他分别是前，左，右。如果它以负速度运动我就在小车的后面防止它。这个类扩展子 sprite 我们想

这将使我们能够直观地看到各点的汽车将在那里接触到的游戏关卡瓷砖，将有助于测试出的碰撞检测。

这是本地的位置

/source/classes/com/efg/framework/LookAheadPoint.as/

下面是完整代码

```
package com.efg.framework
{
import flash.display.*;
/**
 * ...
 * @author Jeff Fulton
 */
public class LookAheadPoint extends Sprite {
public var parentContainer:Sprite;
private var circle:Shape;
private var circleColor:Number;
public function LookAheadPoint(x:Number, y:Number, parentContainer:Sprite, 圆
circleColor:Number=0xffff00) {
THE ESSENTIAL GUIDE TO FLASH GAMES
432
this.x = x;
this.y = y;
this.parentContainer = parentContainer;
this.circleColor = circleColor;
circle = new Shape();
circle.graphics.clear();
circle.graphics.lineStyle(1, circleColor);
circle.graphics.drawCircle(-1, -1, 2);
addChild(circle);
}
```



```
public function show():void {  
    parentContainer.addChild(this);  
}  
public function hide():void {  
    parentContainer.removeChild(this);  
}  
}  
}
```

这个类接收四个参数

界面的 x 坐标

界面的 y 坐标

显示对象的父类添加或移除他自己

LookAheadPoint 实例不需要显示。他们可以打开和关闭调用显示和隐藏方法。

## 创建 CarBlitSprite 类

CarBlitSprite 类是 BlitSprite 的子类在第七章中介绍过。我们增加一些属性。这些将在 DriveSheSaid.as 代码里,稍后介绍。我们覆盖 BlitSprite 的 updateTile 方法。

这里增加的功能让我们的小车通过 tilelist 数组向前向后移动。让小车车轮向反方向移动。

文件在本地结构

/source/classes/com/efg/framework/CarBlitSprite.as

这个类的代码

```
package com.efg.framework  
{  
    import com.efg.framework.BlitSprite;  
    import com.efg.framework.TileSheet;  
    public class CarBlitSprite extends BlitSprite{  
        public var reverse:Boolean = false;  
        public var velocity:Number = 0;  
        public var deceleration:Number = 0;  
        public var acceleration:Number=0;  
        public var maxVelocity:Number=0;  
        public var nextRotation:Number=0;  
        public var turnSpeed:Number = 0;  
        public var maxTurnSpeed:Number;
```



```
public var minTurnSpeed:Number;
public var move:Boolean = false;
public var reverseVelocityModifier:Number = 0;
public var radius:int = 0;
public var worldX:Number;
public var worldY:Number;
public var worldNextX:Number;
public var worldNextY:Number
public function CarBlitSprite(tileSheet:TileSheet, tileList:Array,
firstFrame:int) {
super(tileSheet,tileList,firstFrame);
}
override public function updateCurrentTile():void {
if (animationLoop) {
if (animationCount > animationDelay) {
animationCount = 0;
if (reverse) {
currentTile--
}else {
currentTile++;
}
doCopyPixels = true;
if (currentTile > tileList.length - 1) {
currentTile = 0;
if (useLoopCounter) loopCounter++;
}
if (currentTile < 0) {
currentTile = tileList.length - 1;
if (useLoopCounter) loopCounter++;
}
}
animationCount++;
}
}
}
}
```

让我们讨论一下 updateTile 方法的细节。因为它主要阐述了小车在我们的游戏中如何运动。

1.第一，如果 animationLoop 变量设置为 true，代码

2.下一步，如果 animationCount 大于 animationDelay 我们移动到下一个 tilelist 数组的 tile。如果车向前，我们在 currentFrame 属性上+1。如果向后则-1。



3.Game.as 动态设置 animationDelay 属性。

4.loopCounter 变量，它不用再这个游戏中。

## 创建 Camera2D 类

Camera2D 类不像开始介绍的那样。它可以很方便的给类里封装的变量赋值。这样我就可以保证所有的摄像机相关的设置在一起。

你也许觉得这是一种很奇怪的代码组织方式。对。你可以把这些变量分的很碎，可以根据需要做细微的调整。我们把他们分开我们可以假设在以后可以在子类中进行封装。

向我们这样的老家伙，摄像机类通常用 c 语言设计的很完美。在 90 年代我们创建了一些 DOS 游戏。（它们没有发布，大多数是在学校开始学习编程的时候做的）。我们大部分的编程思想来自于这个时间。C 不是一个面向对象的语言（尽管它的近亲 c++ 是）这个结构是创建一个序列类把一些列的变量放到一个单元中使用。Camera2D 类几乎就是 jeff 以前未完成的 Asteroids-like C 游戏中的相同功能的复制品。当我们搜寻写在书中的事情，我们找到一些更古老的代码，这是创建游戏的灵感我们将在下一章中介绍。

Camera2D 在本地的这里

`/source/classes/com/efg/framework/Camera2D.as`

这里是源代码

```
package com.efg.framework
{
import flash.display.*;
import flash.geom.*;
/**
 * ...
 * @author Jeff Fulton
 */
public class Camera2D {
public var width:int;
public var height:int;
public var cols:int;
public var rows:int;
public var x:Number;
public var y:Number;
public var startX:Number;
public var startY:Number;
public var nextX:Number;
public var nextY:Number;
//the buffer is 1 tile longer and higher than the camera
//well first copy all of the tiles to the buffer
```



```
//then copy just the portion we need to the camera
//buffer
public var bufferBD:BitmapData;
public var bufferWidth:int;
public var bufferHeight:int;
public var bufferRect:Rectangle;
public var bufferPoint:Point;
public function Camera2D() {
}
}
}
```

就像我早先描述的，Camera2D 类是一个简单的 BitmapData canvas 用来保存输出到界面上的数据。xy 坐标表示开始复制 50\*50 的格子，每个格子是 32\*32 像素到缓冲区（每个方向是 1600 像素）。这个缓冲区实际上包含足够的空间存储 13 个 tile。所以缓冲区包含 416 像素（13\*32 = 416）y 方向是一样的。当缓冲区复制的最终数据输出到 canvas 我们开始移动实际的其实像素 xy 仅仅复制 384\*384 像素

## 双缓冲或三缓冲

Camera2D 在游戏中担负着双缓冲显示对象。你可能注意到了我们每帧复制所有的 tile。使用缓冲可以解决屏幕重绘操作造成屏幕的闪烁问题。通过循环所有的 tile 逐个复制像素，每秒 114 个 tile 执行 40 次。

当我们写代码的时候你会看到我们使用另一个缓冲区来显示。你可能在前几章看到过类似的。之前我们复制缓冲区到 canvas，我们调用 canvas 的 lock 方法。这可以阻止输出的 bitmap 实例持有 canvas 直到复制操作结束。我们还可以创建三重缓冲，我们首先复制所有的 tiles 到界面的关闭缓冲中，然后我们锁住 canvas 并且复制需要的像素从相机缓冲区到 canvas 界面。最后我们把解开界面的锁定  
canvasBD.unlock

## 给 Drive She Said 创建特殊的类

Drive She Said 游戏剩下的类和 Main.as 以及 Library.as 在同一个目录下

## TileSheeDataXML 类

在第六章，我们为 No Tanks! 游戏创建了这个类。这个版本我们稍微简化一下。



这是在 flash ide 下的路径

/source/projects/driveshesaid/flashIDE/com/efg/games/driveshesaid/TileSheetDataXML.as

这是在 Flex SDK 下的路径

/source/projects/driveshesaid/flexSDK/src/com/efg/games/driveshesaid/TileSheetDataXML.as

这里是 TileSheetDataXML 的所有代码

```
package com.efg.games.driveshesaid{
public class TilesheetDataXML {
public static var XMLData:XML=
<tilesheet>
<tile id="0" name="player" type="sprite"/>
<tile id="1" name="player" type="sprite"/>
<tile id="2" name="player" type="sprite"/>
<tile id="3" name="heart" type="sprite"/>
<tile id="4" name="clock" type="sprite"/>
<tile id="5" name="skull" type="sprite"/>
<tile id="6" name="wall" type="nonwalkable"/>
<tile id="7" name="wall" type="nonwalkable"/>
<tile id="8" name="wall" type="nonwalkable"/>
<tile id="9" name="wall" type="nonwalkable"/>
<tile id="10" name="wall" type="nonwalkable"/>
<tile id="11" name="wall" type="nonwalkable"/>
<tile id="12" name="wall" type="nonwalkable"/>
<tile id="13" name="wall" type="nonwalkable"/>
<tile id="14" name="wall" type="nonwalkable"/>
<tile id="15" name="wall" type="nonwalkable"/>
<tile id="16" name="wall" type="nonwalkable"/>
<tile id="17" name="wall" type="nonwalkable"/>
<tile id="18" name="wall" type="nonwalkable"/>
<tile id="19" name="wall" type="nonwalkable"/>
<tile id="20" name="wall" type="nonwalkable"/>
<tile id="21" name="road" type="walkable"/>
<tile id="22" name="goal" type="sprite"/>
<tile id="23" name="goal" type="sprite"/>
<tile id="24" name="none" type="nonwalkable"/>
<tile id="25" name="none" type="nonwalkable"/>
<tile id="26" name="none" type="nonwalkable"/>
<tile id="27" name="none" type="nonwalkable"/>
<tile id="28" name="none" type="nonwalkable"/>
<tile id="29" name="none" type="nonwalkable"/>
```



```
</tilesheet>;  
} // end class  
} // end package
```

## Level 类

像第六章的关卡数据一样，我们使用 level 存储数据。我们做这些是保证所有的过渡在一个 swf 文件里面。这是所有游戏关卡的父类

这是在 flash ide 下的路径

/source/projects/driveshesaid/flashIDE/com/efg/games/driveshesaid/Level.as

这是在 Flex SDK 下的路径

/source/projects/driveshesaid/flexSDK/src/com/efg/games/driveshesaid/Level.as

下面是 level 的所有代码

```
package com.efg.games.driveshesaid  
{  
    /**  
     * ...  
     * @author Jeff Fulton  
     */  
    public class Level{  
        public var map:Array;  
        public var backGroundTile:int;  
        public var percentNeeded:Number;  
        public var playerStartFacing:int;  
        public var timerStartSeconds:int;  
        public var heartScore:int;  
        public var clockAdd:int;  
        public var wallAdjust:Number;  
        public var skullAdjust:Number;  
        public var wallDriveColor:uint;  
        public function Level() {  
        }  
    }  
}
```

现在让我们快速转移到 level1.as，使用它定义关卡变量。

译林军书组敬献《Flash 游戏编程指南》



```
}  
}  
}
```

当我们创建自己的数据文件时我们最需要注意的事情是，他们必须和类的结构相匹配，特别是 level 的变量在类的最上边。下面

backGroundTile：这个是 tile 表上的 tile 创建所有透明 tiles 的背景。

percentNeeded:这个是我们一共需要收集的红心数量。

playerStartFacing:小车开始的角度

timerStartSeconds:关卡开始时剩余的时间

heartScore:收集红心获得的分数。

clockAdd:收集一个时钟增加的秒数。

wallAdjust:当玩家撞击墙体时，对于碰撞的反映。这里是让玩家的速度降低的百分比。

skullAdjust:这个变量和 wallAdjust 很像，它表示撞到头骨的反映

wallDriveColor:32 位（因此它包含一个透明通道）表示小车不能驶入的区域。如果

## 迭代 Game 类

我们创建 Game 的一个子类 DriveSheSaid.as。我们快速的迭代出每一个组成我们的游戏。一些代码我们在前几章经常遇到可能已经很熟悉了。另一些可能还比较新。我们，但是我们只停留在讨论新代码上。

这是在 flash ide 下的路径

/source/projects/driveshesaid/flashIDE/com/efg/games/driveshesaid/DriveSheSaid.as

这是在 Flex SDK 下的路径

/source/projects/driveshesaid/flexSDK/src/com/efg/games/driveshesaid/DriveSheSaid.as

## 创建游戏外形，变量，构造方法

在你看完第一段代码后我们讨论下导入的变量

```
package com.efg.games.driveshesaid  
{  
import flash.display.*;  
import flash.geom.Point;
```



```
import flash.geom.Rectangle;
import flash.events.Event;
import flash.utils.getTimer;
import flash.events.KeyboardEvent;
import com.efg.framework.CustomEventSound;
import com.efg.framework.Game;
import com.efg.framework.CustomEventLevelScreenUpdate;
import com.efg.framework.CustomEventScoreBoardUpdate;
import com.efg.framework.CustomEventSound;
import com.efg.framework.BlitSprite;
import com.efg.framework.TileSheet;
import com.efg.framework.Camera2D;
import com.efg.framework.BasicFrameTimer;
import com.efg.framework.LookAheadPoint;
import com.efg.framework.CarBlitSprite;
/**
 *
 * ...
 * @author Jeff Fulton
 */
public class DriveSheSaid extends Game {
    public static const KEY_UP:int = 38;
    public static const KEY_DOWN:int = 40;
    public static const KEY_LEFT:int = 37;
    public static const KEY_RIGHT:int = 39;
    public static const TILE_WALL:int = 0;
    public static const TILE_MOVE:int = 1
    public static const SPRITE_PLAYER:int = 2;
    public static const SPRITE_GOAL:int = 3;
    public static const SPRITE_CLOCK:int = 4;
    public static const SPRITE_SKULL:int = 5;
    public static const SPRITE_HEART:int = 6;
    public static const STATE_SYSTEM_GAMEPLAY:int = 0;
    public static const STATE_SYSTEM_LEVELOUT:int = 1;
    private var systemFunction:Function;
    private var currentSystemState:int;
    private var nextSystemState:int;
    private var lastSystemState:int;
    private var level:int = 0;
    private var levelData:Level;
    private var levels:Array = [undefined,new Level1()];
    //tiles
    private var mapTileWidth:int=32;
```



---

```
private var mapTileHeight:int=32;
//display
private var canvasBitmapData:BitmapData;
private var backgroundBitmapData:BitmapData;
private var canvasBitmap:Bitmap;
private var backgroundBitmap:Bitmap;
//world
private var world:Array=new Array();
private var worldCols:int=50;
private var worldRows:int=50;
private var worldWidth:int=worldCols*mapTileWidth;;
private var worldHeight:int=worldRows*mapTileHeight;
//camera
private var camera:Camera2D = new Camera2D();
//for drawing cameraAreaTiles
private var tileRect:Rectangle;
private var tilePoint:Point;
private var tileSheetData:Array;
//game specific
private var heartsTotal:int = 0;
private var heartsNeeded:int = 0;
private var heartsCollected:int = 0;
private var timeLeft:int = 0;
private var score:int;
private var goalReached:Boolean = false;
//*** level specific ***
private var levelHeartScore:int;
private var levelPlayerStartFacing:int;
private var levelTimerStartSeconds:int;
private var levelSkullAdjust:Number;
private var levelWallAdjust:Number;
private var levelClockAdd:int;
private var levelBackGroundTile:int;
private var levelPrecentNeeded:Number;
private var levelWallDriveColor:Number;
/** player car stuff
private var player:CarBlitSprite;
private var playerFrameList:Array;
private var playerStarted:Boolean = false;
//*sounds
private var carSoundDelayList:Array = [90,80,70,60,50,40,30,20,15,10,0];
private var carSoundTime:int = getTimer();
/** keyboard input
```



```
private var keyPressList:Array = [];  
private var keyListenersInit:Boolean = false;  
/** game loop  
private var gameOver:Boolean = false;  
/** look ahead points  
// The Vector class requires Flash Player 10 publishing  
// It can be swapped with an Array is only Flash Player 9 publishing is available:  
/// private var lookAheadPoints:Array=[];  
private var lookAheadPoints:Vector.<LookAheadPoint>=new Vector.<LookAheadPoint>(3,false);  
/**count down timer  
private var countdownTimer:BasicFrameTimer = new BasicFrameTimer(40);  
//***** Flex *****  
private var tileSheet:TileSheet = new TileSheet(new Library.TileSheetPng().bitmapData, mapTileWidth, mapTileHeight);  
//***** End Flex *****  
//***** Flash IDE *****  
//private var tileSheet:TileSheet = new TileSheet(new TileSheetPng(0,0), tilewidth, tileheight);  
//***** End Flash IDE *****  
public function DriveSheSaid() {  
    init();  
    this.focusRect = false;  
}
```

## 按键常量

我们为键盘上的四个方向键设置一些常量

```
public static const KEY_UP:int = 38;  
public static const KEY_DOWN:int = 40;  
public static const KEY_LEFT:int = 37;  
public static const KEY_RIGHT:int = 39;
```

## 内部状态机制

开始创建两个变量来处理 Game.as 内部的状态机制，它和 Main.as 一样。

```
public static const STATE_SYSTEM_GAMEPLAY:int = 0;  
public static const STATE_SYSTEM_LEVELOUT:int = 1;  
private var systemFunction:Function;  
private var currentSystemState:int;  
private var nextSystemState:int;
```



```
private var lastSystemState:int;
```

这两个状态是为游戏角色设置的

其它四个变量用来处理和操作状态。

## tiles , 显示 , 世界

tiles, display, 和 world 变量和 No Tanks!游戏中的很像, 但在 Drive She Said 中稍作修改。

mapTileHeight 和 mapTileWidth 分别是 tile 的宽和高。在 No Tanks! 被称作 tileWidth 和 tileHeight。这个 display 变量和 No Tanks! 一样。添加一个 Bitmap 实例 ( backgroundBitmap ), 把它放到单独的一个层里面。我们这样做是因为没有必要在摄像机移动或界面重绘是重绘背景。world 数组在 No Tanks! 被叫做 levelTileMap。我们也需要一些新的变量指明 world 的列 , 行 , 宽 , 高。因为它和 canvasBitmapData 不一样。

```
//tiles
private var mapTileWidth:int=32;
private var mapTileHeight:int=32;
//display
private var canvasBitmapData:BitmapData;
private var backgroundBitmapData:BitmapData;
private var canvasBitmap:Bitmap;
private var backgroundBitmap:Bitmap;
//world
private var world:Array=new Array();
private var worldCols:int=50;
private var worldRows:int=50;
private var worldWidth:int=worldCols*mapTileWidth;
private var worldHeight:int=worldRows*mapTileHeight;
```

## 摄像机

camera 是我们早先创建的 Camera2D 类的一个实例。

```
//camera
private var camera:Camera2D = new Camera2D();
```

## 汽车的声音

汽车的声音以特殊的方式处理。根据车速度的不同播放不同的声音。

```
/*sounds
```



```
private var carSoundDelayList:Array = [90,80,70,60,50,40,30,20,15,10,0];  
private var carSoundTime:int = getTimer();
```

carSoundDelay 数组保存了一些毫秒级的声音。例如在速度为 0 是。

## 创建工作类

在这里我们要完善 DriveSheSaid.as 让它们重复执行。建立一些桩函数让它在游戏中执行。当然我们也要设置一些内部状态机。简单点在构造方法里面添加以下信息。

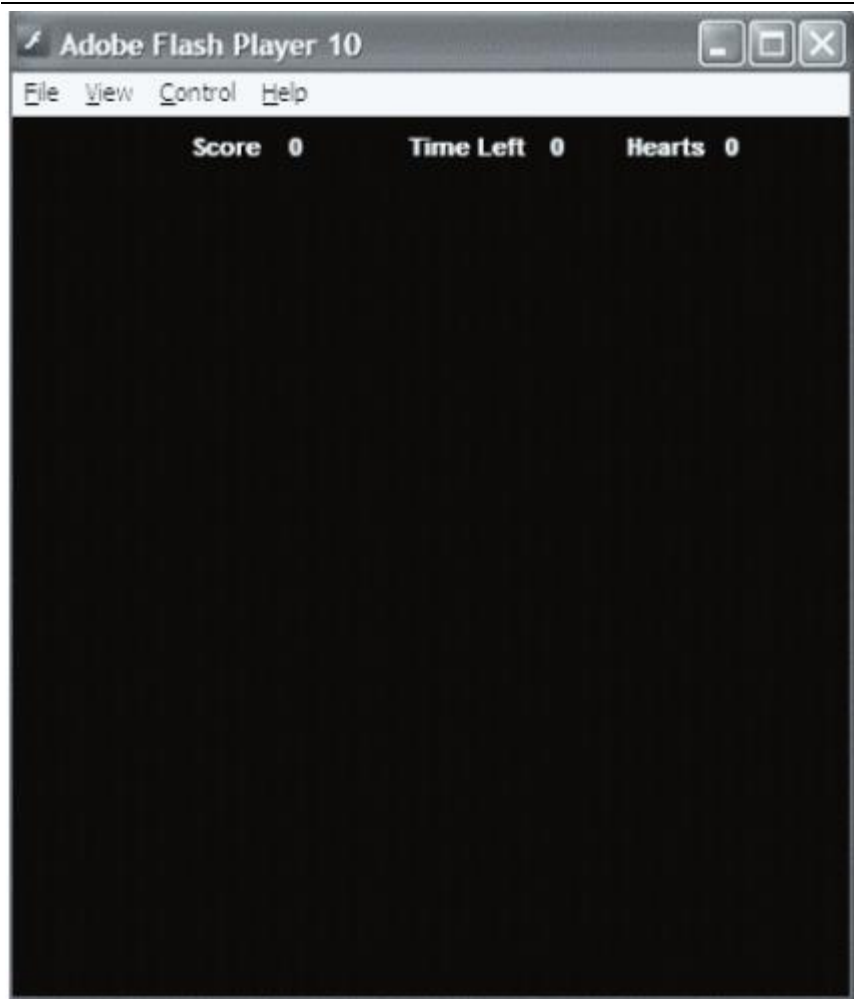
```
private function init():void {}  
override public function newGame():void {  
    switchSystemState( STATE_SYSTEM_GAMEPLAY );  
}  
override public function newLevel():void {  
    stage.focus = this;  
}  
private function restartPlayer():void {}  
private function updateScoreBoard():void {}  
override public function runGame():void {  
    systemFunction();  
}  
public function switchSystemState(stateval:int):void {  
    lastSystemState = currentSystemState;  
    currentSystemState = stateval;  
    switch(stateval) {  
        case STATE_SYSTEM_GAMEPLAY:  
            systemFunction = systemGamePlay;  
            break;  
        case STATE_SYSTEM_LEVELOUT :  
            systemFunction = systemLevelOut;  
            break;  
    }  
}  
private function systemGamePlay():void {}  
private function systemLevelOut():void {}  
private function levelOutComplete():void {}  
private function checkInput():void {}  
private function checkforEndGame():void {}  
private function checkforEndLevel():void {}  
private function update():void {}  
private function checkCollisions():void {}
```



```
private function render():void {}
private function drawCamera():void {}
private function drawPlayer():void {}
private function addToScore(val:Number):void {}
private function initTileSheetData():void {}
private function setupWorld():void {}
private function dispose(object:BlitSprite):void {}
private function disposeAll():void {}
private function timesUpListener(e:Event):void {}
private function keyDownListener(e:KeyboardEvent):void {
    keyPressList[e.keyCode]=true;
}
private function keyUpListener(e:KeyboardEvent):void {
    keyPressList[e.keyCode]=false;
}
}
}
}
```

存根函数的唯一值得注意是游戏内部的状态机。就像 Main 类，systemFunction 方法涉及调用游戏的每个计时器。switchSystemFunction 方法用来改变状态。

如果你运行这段代码，你会在紫色的关卡界面淡出后看到一个全黑的游戏界面。你需要先点击界面上的标题和说明信息。图 10-6 是令人激动的游戏界面



【图 10-6】

## 建立游戏

最后通过迭代把汽车放到界面上。不要认为这一步很简单，实际上完成它需要更多的步骤。

我们初始化新游戏，读取 tile 表的第一关数据，准备开始游戏。

## 初始化函数

The full code for the init function follows:

```
private function init():void {  
    camera.width=384;  
    camera.height=384;  
    camera.cols=12;  
    camera.rows=12;  
    camera.x=0;  
    camera.y=0;
```



```
camera.bufferBD=new BitmapData(camera.width+mapTileWidth,
camera.height+mapTileHeight,true,0x00000000);
camera.bufferRect=new Rectangle(0,0,camera.width,camera.height);
camera.bufferPoint=new Point(0,0);
tileRect=new Rectangle(0,0,mapTileWidth,mapTileHeight);
tilePoint=new Point(0,0);
//canvasBitmap
canvasBitmapData=new BitmapData(camera.width,camera.height,true,0x00000000);
canvasBitmap=new Bitmap(canvasBitmapData);
backgroundBitmapData = new BitmapData(camera.width, camera.height,
false, 0x000000);
backgroundBitmap = new Bitmap(backgroundBitmapData);
addChild(backgroundBitmap);
addChild(canvasBitmap);
//look ahead points
lookAheadPoints[0] = new LookAheadPoint(0, 0, this);
lookAheadPoints[1] = new LookAheadPoint(0, 0, this);
lookAheadPoints[2] = new LookAheadPoint(0, 0, this);
//to show look ahead points in development
lookAheadPoints[0].show();
lookAheadPoints[1].show();
lookAheadPoints[2].show();
}
```

下面详细介绍一下这个方法的重点部分。

初始化 camera 高度和宽度为显示界面的大小 ( 384\*384 ) 设置显示区域的行和列是 ( 12\*12 )

camera.bufferBitmapData 初始化大小要比 canvas 大一个 tile 的高和宽。我们可以一次移动 32 像素，而不是单个像素的移动

camera.bufferRect 和 buffer.point 的初始化。bufferPoint 必须一直保持在 0, 0 位置。因为它始终，bufferRect 的宽高始终是 384，但是 x 和 y 值会从 0-31 之间变化。

tileRect 和 tilePoint 被用来在，tileRect 大小始终是 32\*32 的正方形，但是 tilepoint，x 和 y 变量根据界面缓冲区的值而改变

canvasBitmap 添加游戏的显示列表并且包含了 384\*384 大小的显示数据。  
camera.bufferBitmapData 将在每帧重绘。

backgroundBitmap 添加 canvasBitmap 它包含的 backgroundBitmapData 是 12\*12 大小的 tile ( 关卡 xml )

backgroundBitmap 和 canvasBitmap 添加显示队列



lookAheadPoints 包含了三个 LookAheadPoint 实例。它们都被加到显示队列里面，但是你可以在测试的时候把它们注释掉。它是用 actionscript3 的 vector 类。Vector 类允许开发者创建优化的固定长度单一类型数组。flash 以这种方式可以很好的使用内存和快速的读取数据。如果你用 flash player9，它可以被换为 array。

## newGame 方法

```
override public function newGame():void {
    switchSystemState( STATE_SYSTEM_GAMEPLAY );
    initTileSheetData();
    player = new CarBlitSprite(tileSheet, playerFrameList, 0);
    addChild(player);
    level = 0;
    score = 0;
    gameOver = false;
    dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
    UPDATE_TEXT, Main.SCORE_BOARD_SCORE, String(score)));
    dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
    UPDATE_TEXT,Main.SCORE_BOARD_TIME_LEFT,"00"));
    dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.
    UPDATE_TEXT,Main.SCORE_BOARD_HEARTS,String(0)));
    //key listeners
    if (!keyListenersInit) {
        stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownListener);
        stage.addEventListener(KeyboardEvent.KEY_UP, keyUpListener);
        keyListenersInit = true;
    }
    countdownTimer.addEventListener(BasicFrameTimer.TIME_IS_UP, timesUpListener,
    false, 0, true);
}
```

下面是关于这个方法重点的描述

- 1.首先我们设置内部状态机变量 SYSTEM\_STATE\_GAMEPLAY 来调用 switchSystemState 方法
- 2.我们调用 initTileSheetData，我们将在下一节介绍
- 3.我们通过引用 tilesheet 初始化我们的玩家 CarBlitSprite;playerFrameList 将要设置 initTileSheetData 方法和开始汽车动画的循环。
- 4.在它之后，我们添加玩家到显示队列后，重制分数，关卡和游戏结束变量。
- 5.我们分发三个自定义事件来重置分数面板是新游戏的默认值。
- 6.我们添加关键事件在我们游戏第一次开始的时候，这是因为 Main 方法在舞台初始化好以后，调用游



戏的实例构造方法。我们不能把它们在 init 方法中初始化。我们在这里处理是因为我们的舞台已经初始化好了，而且我们只需要初始化一次。

7.我们给 DriveSheSaid 添加 TIME\_IS\_UP 的监听事件监听 CountdownTimer 类事例

## initTileSheetData 方法

完成 initTileSheetData 方法的代码

```
private function initTileSheetData():void {
    playerFrameList = [];
    tileSheetData = [];
    var numberToPush:int = 99;
    var tileXML:XML = TilesheetDataXML.XMLData;
    var numTiles:int = tileXML.tile.length();
    for (var tileNum:int = 0; tileNum < numTiles; tileNum++) {
        if (String(tileXML.tile[tileNum].@type) == "walkable") {
            numberToPush = TILE_MOVE;
        } else if (String(tileXML.tile[tileNum].@type) == "nonwalkable") {
            numberToPush = TILE_WALL;
        } else if (tileXML.tile[tileNum].@type == "sprite") {
            switch(String(tileXML.tile[tileNum].@name)) {
                case "player":
                    numberToPush = SPRITE_PLAYER;
                    playerFrameList.push(tileNum);
                    break;
                case "goal":
                    numberToPush = SPRITE_GOAL;
                    break;
                case "heart":
                    numberToPush = SPRITE_HEART;
                    break;
                case "skull":
                    numberToPush = SPRITE_SKULL;
                    break;
                case "clock":
                    numberToPush = SPRITE_CLOCK;
                    break;
            }
        }
        tileSheetData.push(numberToPush);
    }
}
```



## 这里是主要点的介绍

这个方法非常简单和 No Tanks!一样。它的职责是读取 TileSheetDataXML 类并放置 tile 在相应的单元里面

1.第一，它初始化 playerFrameList 和 tileSheetData 数组。playerFrameList 数组包括 ids 0, 1, 2。这是头三帧在我们的 tile 表。

2.它通过循环 xml 节点在数组当中。这是 as3xml 的特点。

3.在循环中，我们首先检查 tile 两个基本类型：TILE\_MOVE 或者 TILE\_WALL。如果它一个或者两个，我们放下

4.最终，numberToPush (表现) 添加到 tileSheetData 数组

## newLevel 方法

现在，让我们完成 newLevel 方法

```
override public function newLevel():void {
    stage.focus = this;
    if (level == levels.length-1) level = 0;
    level++;
    heartsTotal = 0;
    heartsNeeded = 0;
    heartsCollected = 0;
    setupWorld();
    countdownTimer.seconds = levelTimerStartSeconds;
    countdownTimer.min = 0;
    goalReached = false;
    dispatchEvent(new CustomEventLevelScreenUpdate(CustomEventLevelScreenUpdate.
UPDATE_TEXT, String(level)));
    dispatchEvent(new CustomEventHeartsNeeded(CustomEventHeartsNeeded.
HEARTS_NEEDED, String(heartsNeeded)));
    restartPlayer();
}
```

## 这个方法重点部分讲解。

The stage.focus = this;确保游戏接受键盘焦点和接受事件而不用用户去单击舞台。

1.我们检查关卡，如果前一个关卡是最后关卡。我们会重新到第一关。



2.我们设置变量 heartsTotal, heartsCollected, heartsNeeded 为 0 , heartsTotal 表示这关的红心总数量。heartsNeeded 是需要收集的 heartsTotal 乘以 levelPercentNeeded 变量。heartsCollected 记录游戏者在这关已经收集的红心数量。

3.下一步我们调用 setupWorld 方法读取并组织 tile 的数据。我们转换一下。

4.我们重制 countdownTimer 设置开始秒数到 levelTimerStartSeconds , 这个计数器的最小值为 0;

5.我们设置 goalReached 为 false。当 heartsCollected 大于 heartsNeeded 时这个变量设置为 true。

6.我们抛出自定义事件 , 来重制积分板的值。

7.我们重新调用 player 方法。

### setUpWorld 方法

setUpWorld 方法是结合 readBackGroundData 和 readSpriteData 在第 6 , 7 章。它读取关卡并且解析它并且创建当前关卡。

```
private function setUpWorld():void {
    world = [];
    var tileNum:int;
    var numberToPush:int;
    levelData = levels[level];
    levelBackGroundTile = levelData.backGroundTile;
    levelTimerStartSeconds = levelData.timerStartSeconds;
    levelHeartScore = levelData.heartScore;
    levelPlayerStartFacing = levelData.playerStartFacing;
    levelSkullAdjust = levelData.skullAdjust;
    levelWallAdjust = levelData.wallAdjust;
    levelClockAdd = levelData.clockAdd;
    levelPercentNeeded = levelData.percentNeeded;
    levelWallDriveColor = levelData.wallDriveColor;
    for (var rowCtr:int=0;rowCtr<worldRows;rowCtr++) {
        var tempArray:Array=new Array();
        for (var colCtr:int = 0; colCtr < worldCols; colCtr++) {
            tileNum = levelData.map[rowCtr][colCtr];
            if (int(tileSheetData[tileNum]) == TILE_WALL ||
                int(tileSheetData[tileNum]) == TILE_MOVE ) {
                numberToPush = tileNum;
            }else {
                switch(tileSheetData[tileNum]) {
                    case SPRITE_PLAYER:
                        numberToPush = levelBackGroundTile;
                        player.worldX = (colCtr * mapTileWidth) + (.5*mapTileWidth);
                        player.worldY = (rowCtr * mapTileHeight) + (.5 * mapTileHeight);
```



```
break;
case SPRITE_HEART:
    numberToPush = tileNum;
    heartsTotal++;
    break;
case SPRITE_SKULL:
    numberToPush = tileNum;
    break;
case SPRITE_GOAL:
    numberToPush = tileNum;
    break;
case SPRITE_CLOCK:
    numberToPush = tileNum;
    break;
}
}
tempArray.push(numberToPush);
}
world.push(tempArray);
}
heartsNeeded = int(heartsTotal * levelPrecentNeeded);
}
```

## 下面是这个方法要点的解释

1.首先，我们重制 world 数组并且初始化本地的变量循环 tiles。tileNum 将存储当前 tile 表中 tile 的数量。numberToPush 变量存储当前 world 存在的当前行和列。大部分的时间，这个数和 tileNum 一样，只是不包括游戏玩家车的 tile。玩家车 tile 会被 levelBackGroundTile 替换。

2.我们使用 levelData = levels[level]设置当前关卡引用的数据

3.我们给我们的 level-specific 变量赋值（在 level 类里面的早先章节）。

4.我们循环 levelData.map 的二维数组的所有节点。

5.如果 tile 是 TILE\_WALL 和 TILE\_MOVE，我们就简单的把 tileNumber 到 tempArray 里面。如果不是我们就。

6.不像 No Tanks!所有的游戏对象都是 as3 的 sprite。只有 player 是现实的对象。所有的其它 sprites 都是简单的 tiles。我们使用 LookAheadPoint 解决碰撞。

7.switch 语法：通过判断 tileSheetData[tileNum]的类型来设置变量。

8.在最后的循环，numberToPush 是插入 tempArray



9.temparray 添加 world 数组创建一个 50\*50 的二维数组

10.最后 heartsTotal 乘以 levelPrecentNeeded 获得 heartsNeeded 关卡数量

## restartPlayer 方法

restartPlayer 方法包含大量的逻辑来让玩家的汽车在界面上正常的位置。玩家的位置非常重要，在我们看完代码后我们来深入的探讨一下。玩家对象需要被放到显示界面上。当我们的玩家对象在界面上移动时，玩家并不移动。而是由摄像机取代。这样让玩家在界面中间，如果玩家在世界的边缘摄像机就不会变化。

当你详查 update 方法时你要仔细观察逻辑，restartPlayer 方法，我们需要简单的认识一下。因为多数的时间，玩家将在界面的旁边当关卡开始的时候。至少一个方向（横向、者垂直方向或者两个方向上）从界面的边缘开始偏移，从中部开始推拉玩家。如果我们不这样做，玩家不可能在正确的 tile 开始。

## 一起移动汽车和游戏世界

当汽车移动的时候，我们想模拟出不同的移动方式。

游戏界面输出（在我们的游戏中称为 canvasBitmapData）是一个 384\*384 的 bitmapdata。

wolrd 是 50\*50tile 或者是 1600\*1600.我们要保证玩家小车在界面的中心.就像我所说的,移动摄像机的位置比移动小车相对简单.但是如何解决小车在界面的边缘的时候.基于此我们放弃控制汽车反方向的影响

让我们的小车在 700\*900 的游戏世界中运行.在这个事例,汽车的 sprite 必须在输出界面 192\*192 的中心位置.摄像机需要偏移 700\*900 来查找更新 x,y 轴坐标。让我们开始

1. 第一，player.worldX 等于 700，player.worldY 等于 900。
2. camera.x = player.worldX-0.5\*camera.width = 508;camera.y = player.worldY - 0.5\*camrea.height = 708;
3. 那么 camrea 从 508 ,708 开始 ,从 x 轴上从 508 到 892 ,从 y 轴上从 708 到 1102。例如图 10-7 描述的那样。



camera 0, 0 on canvas  
camera 508, 708 on world



player at 192, 192 on canvas  
player at 700, 900 on world

图 10-7

当我们从关卡的开始，我们使用非常简单的逻辑来实现。让我们来看一下例子

1. 玩家开始的位置是在第二行，第三列或者是 64x 和 96y。
2. 开始尝试第一次放置摄像机的位置，让玩家在界面的中心。在例子中输出界面区域

$\text{camera.x} = \text{player.worldX} - (.5 * \text{camera.width}) = 64 - 192 = -128$

$\text{camera.y} = \text{player.worldY} - (.5 * \text{camera.height}) \text{ or } 96 - 192 = -86$

摄像机不能在的位置，那么我们需要重新设置到 0, 0  $\text{camera.x} = 0;$

玩家现在必须在开始的位置进行偏移

$\text{player.x} = \text{player.worldX} + \text{camera.x} = 64 + 0 = 64$

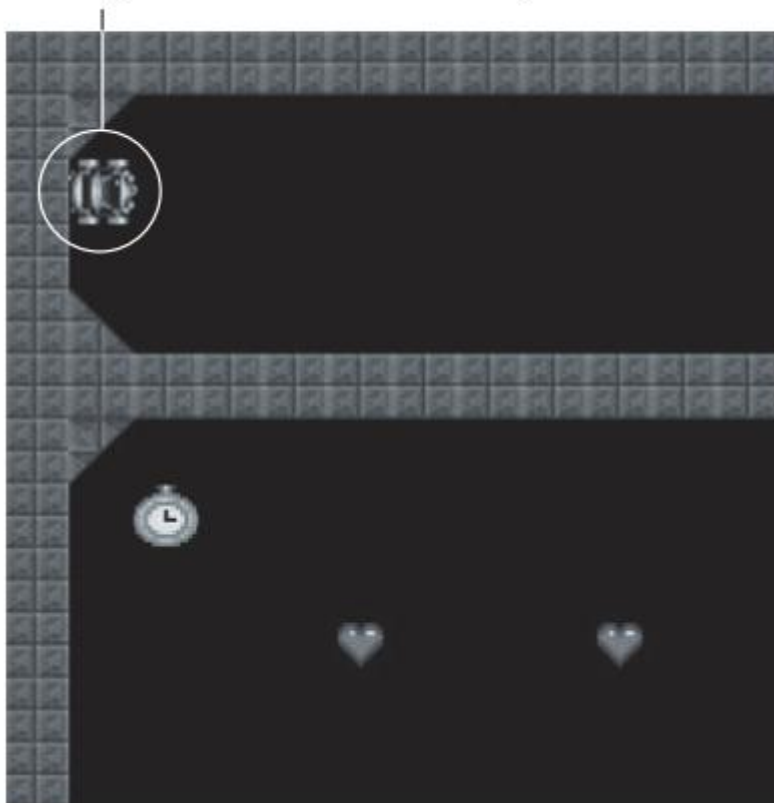
y 轴也一样

camera.y = 0

player.y = playerWorldY + camera.y = 96 + 0 = 96

图 10-8 介绍了这个例子

Player starts at camera position 64, 96  
This happens to be the same world position



```
private function restartPlayer():void {  
    //find the region of the map the player is i  
    camera.x = player.worldX - (.5 * camera.width);  
    camera.y = player.worldY - (.5 * camera.height);  
    if (camera.x < 0) {  
        camera.x = 0;  
        player.x = player.worldX + camera.x;  
    }else if ((camera.x+camera.width) > worldWidth) {  
        camera.x = worldWidth - camera.width;  
        player.x = player.worldX - camera.x;  
    }else {  
        player.x = .5 * camera.width;  
    }  
    if (camera.y < 0) {
```



```
camera.y = 0;
player.y = player.worldY + camera.y;
}else if ((camera.y+camera.height) > worldHeight) {
camera.y = worldHeight - camera.height;
player.y = player.worldY - camera.y;
}else {
player.y = .5 * camera.height;
}
camera.nextX = camera.x;
camera.nextY = camera.y;
player.nextX = player.x;
player.nextY = player.y;
player.worldNextX = player.worldX;
player.worldNextY = player.worldY;
player.dx = 0;
player.dy = 0;
player.nextRotation=levelPlayerStartFacing;
player.turnSpeed = .3;
player.maxTurnSpeed = .6;
player.minTurnSpeed = .3;
player.maxVelocity=10;
player.acceleration =.05;
player.deceleration=.03;
player.radius = .5 * player.width;
player.reverseVelocityModifier = .3;
player.animationDelay = 3;
player.velocity = 0;
//reset Look Aheads
lookAheadPoints[0].x = lookAheadPoints[0].y = 0;
lookAheadPoints[1].x = lookAheadPoints[1].y = 0;
lookAheadPoints[2].x = lookAheadPoints[2].y = 0;
player.visible = false;
drawCamera(); // draw the level so it will roll in from the side
}
```

## 下面是这个方法重点部分的介绍

- 1.我们执行的代码像以前描述的那样：我们设置摄像机的方向所以玩家在界面的中心。
- 2.如果玩家不能在 x,y 轴的中心位置,摄像机就会在相应的轴上为 0,玩家根据相应的变化调整。
- 3.你开始关注使用 nextX 和 nextY 变量.我们经常更新坐标点然后进行碰撞检测并设置 playr 的 x,y 坐标为 nextX,nextY 允许车可以移动的方向。



4. 下一步我们设置小车所有的默认变量。这里主要描述

`player.dx = 0`; `dx` 是沿 `x` 轴移动的速度

`player.dy = 0`; `dy` 是沿 `y` 轴移动的速度

`player.nextRotation = levelPlayerStarFacing`: 它尝试让车转弯

`player.turnSpeed = .3` 当按左键或右键的时候改变旋转角度和速度。

`player.maxTurnSpeed = .6` 这个是速度的最大值。这个值是线型的或者是

`player.minTurnSpeed = .3` 这个是速度的最小值。

`player.maxVelocity = 8`: 这个是

`player.acceleration = .05` 汽车的加速为每一帧向上键的按键时间。

`player.deceleration = .03` 如果汽车没有按向上键如何减速。

`player.reverseVelocityModifier = .3`: 它是速度最大值的百分比。车不能达到他的最大速度除非设置为 1。

`player.radius = .5 * player.width`: 我们使用半径找 `LookAheadPoint` 的位置

`player.animationDelay = 3`: 这个是根据速度进行动画延时调整。它当车的速度过快或过慢都会产生

`player.velocity = 0`: 我们简单设置车的速度为 0

5. 下一步，我们初始化 `LookAheadPoint`

6. 我们在下一步设置玩家不可见，

7. 我们确保当前背景的重绘。

`systemGamePlay` 方法设置状态机 `STATE_SYSTEM_GAMEPLAY` 的状态

```
private function systemGamePlay():void {  
    if (!countDownTimer.started) {  
        countDownTimer.start();  
        playerStarted = true;  
        player.visible = true;  
    }  
    if (playerStarted) {  
        checkInput();  
    }  
    update();  
    checkCollisions();  
    render();  
    checkforEndLevel();  
    checkforEndGame();  
}
```



```
countDownTimer.update();  
updateScoreBoard();  
}
```

## 下面是这个函数要点的详细说明：

- 1.如果 countDownTimer 还没有开始，这个方法必须最先执行。我们要在开始设置玩家的可见状态。
- 2.只有游戏开始以后才接收键盘输入操作。
- 3.我们要贯穿整个游戏的循环过程

## 测试游戏

当我们建立和测试影片的时候，你可以观察关卡界面淡出以及玩家汽车显示在它的起始点，就像 10-9 图所示。你还没有看到关卡，这是因为 drawCamera 方法当前是一个空白。我们要添加 update/render 循环在嵌套 4 中并且在嵌套 4 中结束。



【图 10-9】

添加玩家输入，更新/渲染循环（嵌套 4）

在这个嵌套中，我们最终能看到活动的东西了。当我们完成的时候。你可以驾驶你的小车在游戏世界



里任意驰骋，只是现在还没有对碰撞进行检测。

## 更新方法

这个更新方法包含了大量非常重要的逻辑模拟小车的运动。

```
private function update():void {
    player.move = true;
    /*** update turnSpeed based on velocity
    if (player.velocity == 0) {
        player.turnSpeed = 0;
    }else {
        player.turnSpeed = player.minTurnSpeed + (Math.abs(player.velocity/10));
        if (player.turnSpeed > player.maxTurnSpeed) {
            player.turnSpeed = player.maxTurnSpeed;
        }
    }
    player.rotation=player.nextRotation;
    var carRadians:Number = (player.nextRotation / 360) * (2.0 * Math.PI);
    var lookRadians:Number;
    player.dx=Math.cos(carRadians)*player.velocity;
    player.dy=Math.sin(carRadians)*player.velocity;
    player.worldNextX += player.dx;
    player.worldNextY += player.dy;
    camera.nextX = player.worldNextX - (camera.width * .5);
    camera.nextY = player.worldNextY - (camera.height * .5);
    if (camera.nextX <0) {
        camera.nextX = 0;
        player.nextX += player.dx;
    }else if (camera.nextX > (worldWidth - camera.width - 1)) {
        camera.nextX = (worldWidth - camera.width - 1);
        player.nextX += player.dx;
    }
    if (camera.nextY <0) {
        camera.nextY = 0;
        player.nextY += player.dy;
    }else if (camera.nextY > (worldHeight - (camera.height - 1))) {
        camera.nextY = (worldHeight - (camera.height - 1));
        player.nextY += player.dy;
    }
    if (player.velocity < 0) {
```



```
player.reverse = true;
lookRadians = ((player.nextRotation-45) / 360) * (2.0 * Math.PI);
lookAheadPoints[0].x=(player.nextX-Math.cos(lookRadians)*player.radius);
lookAheadPoints[0].y = (player.nextY - Math.sin(lookRadians) * player.radius);
//lookRadians the same as carRadians for middle
lookAheadPoints[1].x=player.nextX-Math.cos(carRadians)*player.radius;
lookAheadPoints[1].y = player.nextY - Math.sin(carRadians) * player.radius;
lookRadians = ((player.nextRotation+45) / 360) * (2.0 * Math.PI);
lookAheadPoints[2].x = (player.nextX - Math.cos(lookRadians) * player.radius);
lookAheadPoints[2].y = (player.nextY - Math.sin(lookRadians) * player.radius);
}else {
player.reverse = false;
lookRadians = ((player.nextRotation-45) / 360) * (2.0 * Math.PI);
lookAheadPoints[0].x=(player.nextX+Math.cos(lookRadians)*player.radius);
lookAheadPoints[0].y = (player.nextY + Math.sin(lookRadians) * player.radius);
//lookRadians the same as carRadians for middle
lookAheadPoints[1].x=player.nextX+Math.cos(carRadians)*player.radius;
lookAheadPoints[1].y = player.nextY + Math.sin(carRadians) * player.radius;
lookRadians = ((player.nextRotation+45) / 360) * (2.0 * Math.PI);
lookAheadPoints[2].x = (player.nextX + Math.cos(lookRadians) * player.radius);
lookAheadPoints[2].y = (player.nextY + Math.sin(lookRadians) * player.radius);
}
}
```

## 下面是这个函数要点的详细说明：

1.player.move 是一个设置为 true 的 Boolean 变量。如果我们的三个点在碰到墙体时返回为 true 时我们会认为发生了碰撞，这样我们会设置这个变量为 false。

2.下一步，我们改变玩家的速度。这将直接涉及到车的实际速度，maximum 转速是.6，minimum 是.3，我们可以设置速度为一个简单的线性关系。我们可以简单的把速度以 10 为单位添加给 player.minTurnSpeed。然后我们检查确保不会大于 player.maxTurnSpeed。

3.在本章上一节提到的汽车移动（“看给我们的车添加物理属性”），我们计算新的 dx 和 dy 值使用 player.nextRotation 值和当前玩家的速度。我们用 dx、dy 的值更新 player.worldNextX 和 player.worldNextY 的值。camera.nextX 和 camera.nexty 的值则是根据玩家在世界的相对位置确定的。

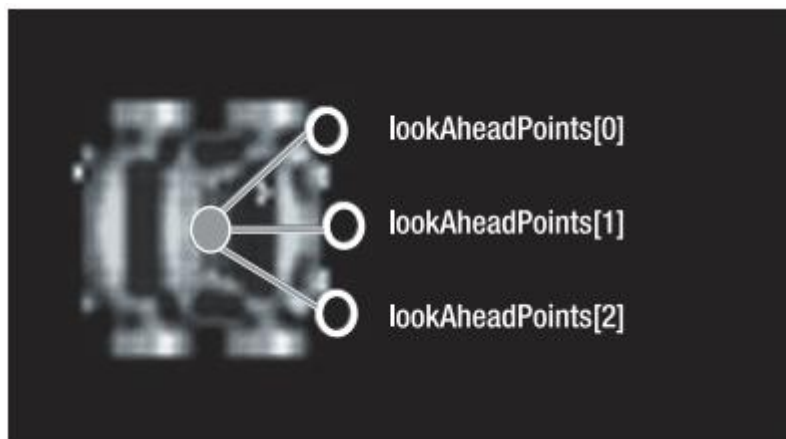
4.下一步 我们设置 camera 的 nextX 和我们在嵌套 2 里面讨论的方向处理很相似。如果 camera.nextX 和 camera.nextY 离开了界面的上下左右，我们会在对应的轴向上回到 0 并且移动玩家替代。

5.最后的事情，如果车向前（速度大于 0）或者车的速度小于 0 时向后，我们在车的前中左右设置三个 look ahead 点。

我们计算以车中心正方向或负方向为 45 角的弧度。在正反两面，以车为中心准确计算 45 度角，再乘



以之前计算过的弧度数据，就可以准得到精确的数值。中间的 LookAheadPoint 使用简单的弧度旋转。



【图 10-10】

我们在图 10-10 绘制的 45 度角不是很精确，但是他们基本证明了 Look ahead 点在玩家对象的 0 角度。

## checkInput 方法

checkInput 方法在每帧根据 STATE\_SYSTEM\_GAMEPLAY 状态而被调用。所有按下的键值被存到 keyPressList 数组中

```
private function checkInput():void {  
    if (keyPressList[KEY_UP]){  
        player.velocity+=player.acceleration;  
        if (player.velocity >player.maxVelocity) player.velocity=player.maxVelocity;}  
        if (!keyPressList[KEY_UP] && player.velocity >0) {  
            player.velocity-=player.deceleration;  
            if (player.velocity <0) player.velocity=0;  
        }  
        if (keyPressList[KEY_DOWN]){  
            player.velocity-=player.acceleration;  
            if (player.velocity < -player.maxVelocity*player.reverseVelocityModifier)②  
            player.velocity = -player.maxVelocity*player.reverseVelocityModifier;  
        }  
        if (!keyPressList[KEY_DOWN] && player.velocity <0) {  
            player.velocity+=player.deceleration;  
            if (player.velocity > 0) player.velocity = 0;  
        }  
    }
```



```
if (keyPressList[KEY_LEFT]){  
    player.nextRotation-=(player.velocity)*player.turnSpeed;  
}  
if (keyPressList[KEY_RIGHT]){  
    player.nextRotation+=(player.velocity)*player.turnSpeed;  
}  
}
```

## 下面是这个方法要点的详细说明：

如果向上键被按下，我们增加(player.acceleration)的速度直到速度的最大值。速度变量保存当前车的速度。他不是真正的矢向量。

如果向上键没有被按下并且速度不是 0，则我们逐渐减少速度值。

如果向下键被按，我们减少加速值直到速度达到最小值。（速度\* reverseVelocityModifier）

如果速度小于 0 并且向下键没有按下，我们增加减速直到速度变为 0。

如果向左向右的按键被按下，我们计算旋转加玩家速度。我们向左加向右减。

## render 方法

render 过程分为两部分：

渲染界面和渲染玩家小车。render 方法的调用。

```
private function render():void {  
    drawCamera();  
    drawPlayer();  
}  
private function drawCamera():void {  
    //calculate starting tile position  
    if (player.move) {  
        camera.x = camera.nextX;  
        camera.y = camera.nextY;  
    }  
    //find starting tiles  
    var tileCol:int=int(camera.x/mapTileWidth);  
    var tileRow:int = int(camera.y / mapTileHeight);  
    var rowCtr:int=0;  
    var colCtr:int=0;
```



```
var tileNum:int;
//camera buffer is 1 tile larger (51 rows).
//For last tile row, make sure to only copy 50
//here we simply catch the exception
for (rowCtr = 0; rowCtr <= camera.rows; rowCtr++) {
    if (rowCtr+tileRow == worldRows) break;
    for (colCtr = 0; colCtr <= camera.cols; colCtr++) {
        if (colCtr + tileCol ==worldCols ) break;
        tileNum = world[rowCtr + tileRow][colCtr + tileCol];
        tilePoint.x=colCtr*mapTileWidth;
        tilePoint.y=rowCtr*mapTileHeight;
        tileRect.x = int((tileNum % tileSheet.tilesPerRow))*mapTileWidth;
        tileRect.y = int((tileNum / tileSheet.tilesPerRow)) * mapTileHeight;
        camera.bufferBD.copyPixels(tileSheet.sourceBitmapData, ②
        tileRect, tilePoint);
    }
}
//put buffer rect in corrct position what pixel
//to start the copy on in the left-hand top tile
camera.bufferRect.x=camera.x % mapTileWidth;
camera.bufferRect.y = camera.y % mapTileHeight;
//trace(bufferRect.x + "," + bufferRect.y);
canvasBitmapData.lock();
canvasBitmapData.copyPixels(camera.bufferBD,camera.bufferRect,②
camera.bufferPoint);
canvasBitmapData.unlock();
}
private function drawPlayer():void {
    if (player.velocity == 0) {
        player.move = false;
    }
    if (player.move) {
        player.animationDelay = player.maxVelocity - player.velocity;
        player.animationLoop = true;
        player.updateCurrentTile();
        player.renderCurrentTile();
        player.x = player.nextX;
        player.y = player.nextY;
        player.worldX = player.worldNextX;
        player.worldY = player.worldNextY;
    }else {
        player.animationLoop = false;
    }
}
```



```
if (getTimer() > carSoundTime + 1)
{
    carSoundDelayList[Math.abs(int(player.velocity)))] {
    dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, 1,
    Main.SOUND_CAR, false, 1, 0));
    carSoundTime = getTimer();
}
}
```

## drawCamera 方法

drawCamera 方法是界面渲染的核心。这个过程我们需要每帧都要做以下的操作。

- 1.更新摄像机的起始位置
- 2.查找摄像机的开始 tile 所在的行和列。
- 3.循环复制这些行和列到缓冲区
- 4.由缓冲区绘制界面

如果 player.move 变量 ( 当 checkCollisions 判断玩家撞到墙的时候返回 false ) 是 true 我们更新 camera.x 和 camera.y 为相应的 nextX 和 nextY。

下一步我们处理 tileCol 和 tileRow 变量。它们表示开始复制行和列从 world 数组到 camera.bufferBitmapData 中。

现在我们循环 13 行 13 列中的 tile 把他们绘制到缓冲区 ,13 行开始于 tileRow 经过 tileCol 加 12。13 列开始于 tileCol 经过 tileCol 加 12。

注意 rowCtr 和 colCtr 不能超过 50 ( 也就是(worldRows 和 worldCols 的值 ) 我们这样做是因为当我们到达界面边缘的时候只有 12 行或 12 列在世界中 ,我们不能建立一个 13 行或 13 列的缓冲区在靠近边缘的方向上。在这个实例中我们打断了相应的循环。

下一步 ,我们重绘 tile 和 tile 表的矩形范围。最后我们通过调用 camera.bufferBitmapData 以像素级的方式重绘 tile

一旦 bufferBitmapData 完成 , 我们设置 camer.bufferRect 的矩形范围从第一个 tile 开始。x 值是 camera.x / mapTileWidth 的余数。y 值也是 camera.y / mapTileHeight 的余数。缓冲区已经被设置为大小 384\*384 不用去改变 , 最后我们锁定 canvasBitmapData 并且复制 bufferBitmapData 384\*384 给 canvasBitmapData 从 0 , 0 开始。



## drawPlayer 方法

drawPlayer 方法是玩家功能渲染的核心。需要关注选择绘制正确的玩家汽车动画和根据速度播放相应的声音。

- 1.如果玩家没有停止，我们让玩家保持动画状态（改变轮子不同帧的动画）。
- 2.如果玩家动画正确，更新玩家动画倒转它的速度。（好的速度低于或者在两帧中延时）。
- 3.从 tilesheet 里更新玩家当前的 tile。
- 4.渲染玩家当前的 tile 到 player.bitmapData.
- 5.设置玩家新的 x , y 轴。我们更新 player.worldX 和 player.worldY 的值。
- 6.播放小车声音的快慢是根据新的速度。

在 checkCollisions 方法中，我们根据是否和墙体碰撞来判断设置 player.move 的属性是 true 还是 false。如果我玩家可以移动，我们就更新玩家的属性。

animationDelay 是一系列的帧，从 tile 表里面选出来替换玩家 tile。在 tile 列表中有三个不同的小车图像，根据小车向前向后模拟

相对速度。因为真的速度和数量都是线性的，所以我们可以使用速度来计算延迟。

只有玩家被允许移动，我们才能更新 nextX 和 nextY 的值。注意如果玩家不在界面的边缘，nextX 和 nextY 的值就和 x , y 的值一样。这是因为我们想让玩家始终停留在场景的中心位置除非界面停止滑动。

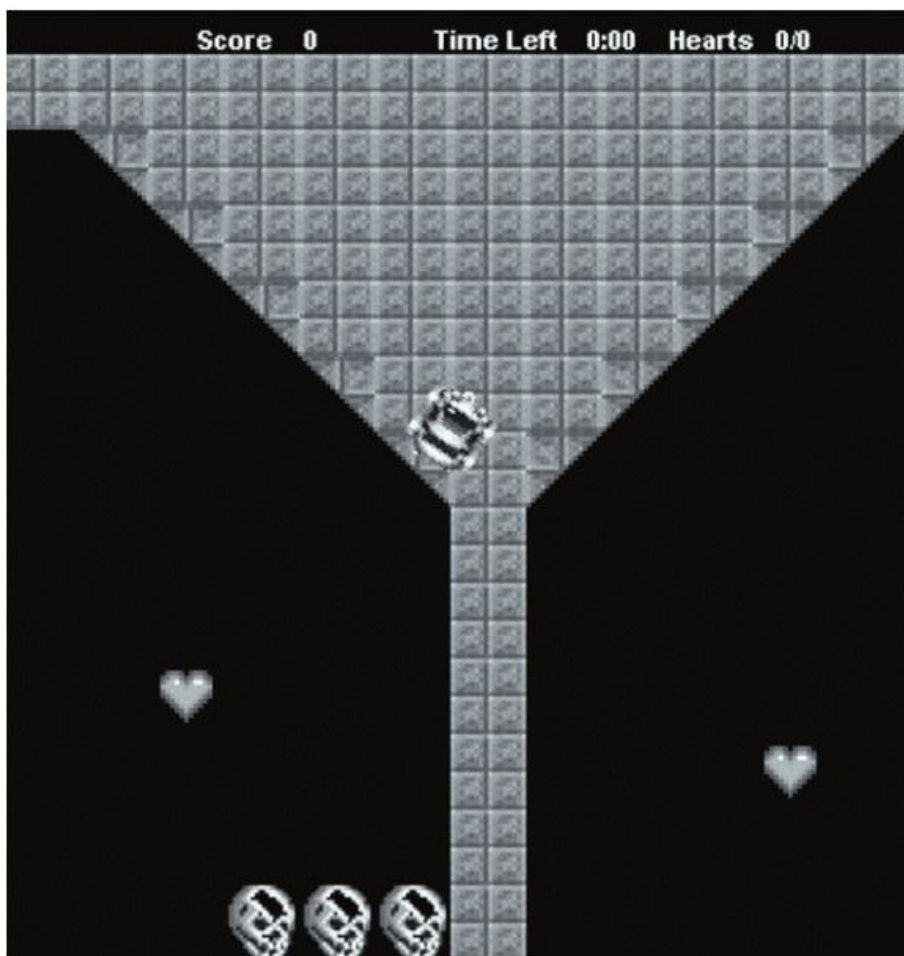
玩家汽车的声音播放是根据一个数组。我们早先定义过这个数组

```
private var carSoundDelayList:Array = [90,80,70,60,50,40,20,10,0];
```

放入数组中的值是基于玩家速度。这类似于如何让车轮动画延迟。我们需要做大量的测试才能测出我们想要使用的值

## 测试嵌套 4

当我们建立一个测试影片后，我们会看到关卡界面会淡出，玩家汽车会在开始的位置。你需要单击一下界面才能用方向键控制汽车。你可以在游戏世界中任意移动没有任何碰撞检测。



【图 10-11】

碰撞检测和最后的关卡或者是游戏

在这个嵌套中，我们将完成剩下的游戏逻辑，在嵌套 4，我们完成了更新和渲染方法。小车现在可以在场景中行驶，但是还没有做墙，红心，或者其它 sprite 的影响。

Drive She Said 使用三个 LookAheadPoint 对象来检测是否他们碰撞。他们定位点来表示小车的下一个位置。我们三个这样可以确保把整个小车覆盖上，以完成检查是否碰撞到其它对象。我们根据 player.nextX 和 player.nextY 更新方法值检查每个 tile 的类型。我们也设置 player.move 变量为 true 在 update 方法中。如果任何一个点碰撞了强，这个变量会被设置成 false。



如果我们用其它的方式处理碰撞，如首先设置 player.move 为 false 然后如果任何一个 LookAheadPoints 没有碰到墙体我们把变量设置为 true。我们可以直观的判断当前点是否在墙上。

## checkCollisions 方法

```
private function checkCollisions():void {
    var lookAheadLength:int = lookAheadPoints.length;
    var row:int;
    var col:int;
    var tileType:int;
    //loop through all three look ahead points
    for (var ctr:int = 0; ctr < lookAheadLength; ctr++) {
        row = (lookAheadPoints[ctr].y+camera.y) / mapTileHeight;
        col = (lookAheadPoints[ctr].x + camera.x) / mapTileWidth;
        tileType = tileSheetData[world[row][col]];
        switch(tileType) {
            case TILE_MOVE:
                //do not need to do anything
                break;
            case TILE_WALL:
                if (canvasBitmapData.getPixel(lookAheadPoints[ctr].x, ②
lookAheadPoints[ctr].y) != levelWallDriveColor) {
                    if (Math.abs(player.velocity) > 1) {
                        //don't keep on playing sound if close to a wall
                        dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,②
Main.SOUND_HIT_WALL, false, 1, 0));
                    }
                    //check for stuck cars=
                    if (player.reverse && player.velocity > .1) {
                        player.velocity = -1;
                    }else if (!player.reverse && player.velocity < .1){
                        player.velocity = 1;
                    }
                    trace("player.velocity=" + player.velocity);
                    player.velocity *= -levelWallAdjust;
                    player.dx *= -levelWallAdjust;
                    player.dy *= -levelWallAdjust;
                    player.move = false;
                }
                break;
            case SPRITE_SKULL:
                player.velocity *= -levelSkullAdjust;
```



```
player.dx *= -levelSkullAdjust;
player.dy *= -levelSkullAdjust;
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, 0,
Main.SOUND_SKULL_HIT, false, 1, 0));
break;
case SPRITE_CLOCK:
countDownTimer.seconds += levelClockAdd;
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, 0,
Main.SOUND_CLOCK_PICKUP, false, 1, 0));
world[row][col] = levelBackGroundTile ;
break;
case SPRITE_HEART:
heartsCollected++;
score += levelHeartScore;
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, 0,
Main.SOUND_HEART_PICKUP, false, 1, 0));
world[row][col] = levelBackGroundTile ;
break;
case SPRITE_GOAL:
if (heartsCollected >= heartsNeeded ) {
goalReached = true;
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, 0,
Main.SOUND_LEVEL_COMPLETE, false, 1, 0));
break;
}
}
}
```

基于此类的碰撞检测我们会在循环中不断的检测三个 LookAheadPoint 是否发生碰撞。

1.如果 tile 的类型是 TILE\_MOVE , 我们什么也不做。这部分的开关

2.如果 tile 的类型是 TILE\_WALL , 我们在第二秒检查该像素点的颜色是否是 levelWallDriveColor 如果是, 我们执行墙体响应代码。

如果玩家的速度大于 1 , 我们播放碰撞墙体的声音。我们仅想在撞击的一瞬间做这个事情 , 所以这段代码必须保证下一阵的速度小于 1。我们用 Math.abs 来获得绝对值。

在车冲撞墙体减速反弹和停止 , 我们确保玩家的车在正确的方向上。下一段代码是让玩家离开墙体。

为了确保玩家在下一帧的速度小于 1。我们添加反方向的 levelWallAdjust 给 player.velocity,play.velocity,player.dy 让玩家

小车向反方向移动的效果。



我们确保 player.move 变量设置为 false。下一帧玩家反方向移动。

3.如果碰撞的是骷髅头我们添加反向的 levelSkullAdjust 变量给 player.dx,player.dy 和 player.velocity 和撞墙的方式相同。我们会用背景 tile 和替换 tile。它在也不会出现在界面上并且播放碰撞骷髅头的声音。

4.如果玩家碰撞了一个时钟，我们添加 countdownTimer 时间：countdownTimer.seconds += levelClockAdd。我们也用背景替换当前的 tile 所以它不会再在界面上出现了。并播放碰撞时钟的声音。

5.如果玩家碰撞了红心，我们添加 heartsCollected 变量给总分。我们也把它替换为背景 tile，所以它再也不会出现在界面上出现，并播放碰撞红心的声音。

6.如果玩家碰撞了终点 sprite 并且收集了足够的红心，goalReached 变量会被赋值为 true。我们也会播放到达目标的声音。

## 游戏剩下的方法

最后的方法没有添加新的游戏逻辑，唯一新的内容是 systemLevelOut 方法使用 STATE\_SYSTEM\_LEVELOUT 状态，goalReached 变量设置为 true。

添加剩余的方法给 DriveSheSaid.as 文件，最后编译这个类包并运行最终的游戏。

## 首先我们添加最终的 addToScore 方法

```
private function addToScore(val:Number):void {  
    score += val;  
}
```

## 下一步添加释放方法：

```
private function dispose(object:BlitSprite):void {  
    object.dispose();  
    removeChild(object);  
}
```

## 现在释放了所有的方法：

```
private function disposeAll():void {  
    if (gameOver) {  
        countdownTimer.removeEventListener(BasicFrameTimer.TIME_IS_UP, timesUpListener);  
        dispose(player);  
    }  
}
```

当 CountdownTimer 实例为 0 时表示着游戏结束了，timesUpListener 会被移除。现在添加他

```
private function timesUpListener(e:Event):void {
```



```
countDownTimer.stop();
gameOver = true;
}
```

## checkforEndGame 方法的完整内容

```
private function checkforEndGame():void {
    if (gameOver ) {
        dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, 2,
        Main.SOUND_GAME_LOST, false, 1, 0));
        dispatchEvent(new Event(GAME_OVER));
        countDownTimer.stop();
        disposeAll();
    }
}
```

## 现在我们添加 checkForEndLevel 方法

```
private function checkforEndLevel():void {
    if (goalReached) {
        disposeAll();
        switchSystemState( STATE_SYSTEM_LEVELOUT );
        countDownTimer.stop();
    }
}
```

systemLevelOut 方法移动 DriveSheSaid.as 直到它要移出显示舞台（x 值是 404），这是在调用 levelOutComplete 方法。

```
private function systemLevelOut():void {
    this.x += 4;
    if (this.x >= 404) {
        this.x = 404;
        levelOutComplete();
    }
}
```

## 也需要添加 levelOutComplete 方法

```
private function levelOutComplete():void {
    dispatchEvent(new Event(NEW_LEVEL));
    switchSystemState( STATE_SYSTEM_GAMEPLAY );
}
```

## 最后添加 updateScoreBoard 方法

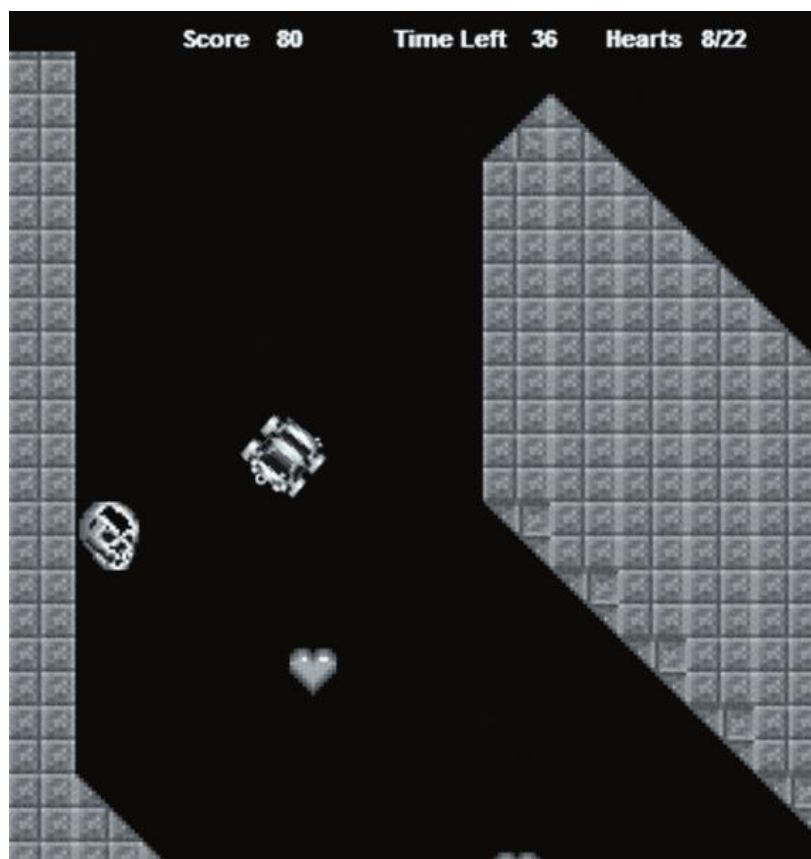
```
private function updateScoreBoard():void {
    dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.2
```



```
UPDATE_TEXT, Main.SCORE_BOARD_SCORE, String(score));  
dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.  
UPDATE_TEXT, Main.SCORE_BOARD_TIME_LEFT, String(timeLeft) ));  
dispatchEvent(new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.  
UPDATE_TEXT, Main.SCORE_BOARD_HEARTS, String(heartsCollected) + "/" +  
String(heartsNeeded))));  
timeLeft = countdownTimer.seconds;  
}
```

## 最终游戏的测试

你现在有一关游戏可以玩了，像图 10-12.当我们测试它，你可以在环境中移动，收集红心和时钟。避免装上骷髅和墙。你需要在时间结束之前到达终点线，并收集足够的红心。



【图 10-12】

## 扩展游戏

再下一章，我们可以添加暂停和静音方法。当然也可以给游戏做一些扩充。在不同的关卡里面收集更多的东西和新的墙体。更多危险和增速的驾驶来提高游戏的乐趣扩展可玩性。



现在你要做的事情是创建不同的关卡。

1.用 Mappy 为关卡创建 xml。

2.备份 Level1.as 重命名为 Level2.as

3 用一个新的二维数组替换第一关的地图变量。用 Mappy 的自定义菜单中的 Export to ActionScript 选项导出关卡 2。

4 制作特别的关卡文件

5.确保在 DriveSheSaid.as 中添加 Level2 的变量定义

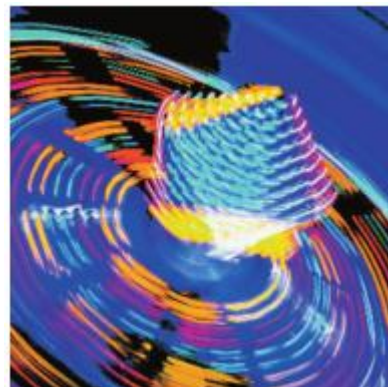
```
private var levels:Array = [undefined, new Level1(), new Level2()];
```

6.重复该过程创建你的关卡。

## 总结

我们在这章中涉及了很多东西，我们快速的完成了它。创建这个游戏我们扩展了 BlitSprite 类是用了 blitting 技术。我们还通过演变书中车的移动创建 tile 世界。我们使用常规的碰撞，也通过颜色的区分扩展了碰撞。游戏演变的高级界面渲染代码和碰撞检测可以 360 度的移动。我们使用三个 look-ahead 点通过车的前后 tile 的像素颜色来判断碰撞。我们也把声音移到自定义的类里面叫做 Library ( 仅仅在 flex sdk 中使用 ) 绑定 mp3 文件而不是 wav 文件。

在下一章，我们将创建这本书中最优化的游戏。炸弹布雷是几何战争，爆炸包含了游戏 timer 循环的优化和对象池以及粒子爆炸



## 第十一章

# 创建一个优化的 Post-Retro 游戏

在这一章中，我们将会创建一个 post-retro 游戏。这个类型的现代游戏运用了经典的 retro 思路并且根据现在的时代进行了改进。两个好的例子，Bit Trip Beat<sup>1</sup>和 Geometry Wars。在 Bit Trip Beat 里，经典的 Pong 游戏被复刻并且融进了现代的节奏音类乐玩法，创造出了一个当代的经典范例。在 Geometry Wars 中，经典的游戏 Asteroids<sup>2</sup>被赋予了一个新的外观，一系列新的操控，一个更大的可滚动的游戏区域，和令人激动的音乐，创造了一个和经典游戏一样让现在的玩家享受的游戏。

## 理解 post-retro 游戏

许多人有这样的感觉，现在 retro 概念的游戏消失了。这并不意味着没有受到 retro 启发的游戏出现，但是这意味着“retro”已经被融合进了主流游戏里并且真的不再单独存在一个这样的游戏类别了。Retro 就是主流。在 2009 年的 E3 上，很多游戏面临窘境或者被解散，同样的 retro 游戏这些年却在迎头赶上（例如：超级马里奥 Wii 版，Nostalgia，和 最终幻想 7）这些只是游戏（注意，好游戏）并且只是因为一些表明的原因才冠以“retro”之名。从某种意义上说，这些有着经典玩法的经典游戏，不一定需要一个 retro aesthetic。

于此同时，也有一些非常健康的老游戏社区：在老系统和模拟器上玩或者专门为一些老硬件写游戏。这是真正复古行为并且应该如此。

<sup>1</sup> Gaijin Games 的音乐节奏类游戏，用音乐游戏的方式来玩 Pong 弹球

<sup>2</sup> 1979 年 由 Ataric Inc. 发行的一款视频电子游戏



尽管如此，一个新的游戏设计趋势正在出现，用现在被接受的游戏类型很难描述(例如 casual, core, retro, viral, and mobile).这个新的趋势由一些新概念构成，以复古的样子出现，但却不是经典老游戏的复刻版。

不是用新的画面重新混合，改编，翻新老的游戏或者老的收藏。也不是在为了老平台上跑的自制游戏。这些游戏是一些完全不同的东西 我们称他们构成的这一类为 “post-retro ”

## 定义 post-retro

Post-retro 游戏是利用复古的美学混合复古和现代的游戏娱乐元素去创造一个全新的体验。

“post-retro ”跟“后现代”有关，这些游戏有老游戏时期的纯怀旧元素并且用复古作为一种新点子的平台。

## 探索后复古游戏的特征

后复古游戏能够并且已经被以多种方式实现，从这个角度来说，这个游戏类别的界限有点模糊。尽管如此，很多这类游戏有一些普遍的特征。不是这个类别所有的游戏都拥有这些列出来的全部特性，但是他们足够的特质，可以被认为是属于这个类别的。

玩起来复古并且现代：这些游戏有着复古感觉的玩法但是同时也感觉不复古（一个悖论）。这种感觉就是能够在以前制作出这种游戏但是没有做，有可能是因为游戏设计的概念经过这些时候已经发展。

复古美：有代表性的，这些游戏利用了一个复古的图形外观(经常是 8 比特，16 比特，或者矢量)并且声音效果也类似的。

现代音乐：音乐是使这些游戏区别于复刻游戏和直接复古游戏的一个有趣的因素。大部分黄金时代游戏机 ( consoles ) 原作几乎没有音乐 ( 有些电脑游戏有，但是那是另一个故事 )。同时很多的这类游戏保持老游戏那种 8 比特或者 16 比特的外观。相对的，无声的背景通常被替换成了迷幻或者催眠音乐。声音效果任然有，但是他们被音效混合到了音乐里面，而不是单独存在。音效通常和游戏的复古外观风格一致。

催眠状态：许多这类的游戏拥有的另一个特性就是加速，大部分的流畅的玩法。在大多数游戏里，生命和分数仍然很重要，但是你可以获得的生命比你再传统的复古游戏里要多。从某种意义上说，相对约束诸多的类复古游戏，后复古类更多的是体验一种后复古玩法的感觉。

背景图片：移动背景图片

粒子效果：经常出现大量的粒子效果。玩法上没有修改太多，但是增加了经典里没有必要出现在游戏里的力学上的和混乱的感觉。

看在怀旧的面子上不怀旧：虽然这些游戏能激发一些玩家的怀旧之情，他们却是被设计来用复古或者怀旧之情进行一些新的体验。

不是复刻:这类游戏不包括简单的复古重制版。如果后复古游戏是一个老游戏改变玩法的版本，这个很重要，那么我们就认为它属于这一这个类别。

## 新兴的后复古游戏的特征。

后复古游戏的特征有一些已经很明显了，但是现在还是缺乏普遍性。



自动射击或移动：这个特征是全新的，并且为了让玩家能更集中在这个新特征上，他们去掉了一些经典玩法的笨重的地方。

回到起点：这是另一个全新的概念，并且涉及到用老游戏的概念作为一个平台来创造一个全新的游戏。我们不想听起来高人一等，但是某些情况下，这些游戏有一种文艺上的感觉。在一些游戏里，游戏的整个思想被撕碎并且反转或者赋予多种含义，这里就会让人产生这种感觉类似于文学中的解构。就像是比喻手法，表面的意思跟他真正的含义完全不一样，甚至是超越的，远远超出了最初设计时候的目的。

## 追溯后复古游戏的历史

虽然后复古游戏的历史还没有被充分的考证，在 1993 年的 Jeff Minter 为 Atari Jaguar 主机开发的 Tempest 2000 游戏里就找到了大部分后复古游戏的特点。

看着这款游戏的内容就像从 1993 年到未来的缩影。Minter 的游戏囊括了几乎所有的后复古类游戏的方向。在游戏玩法上十分类似经典 Tempest 游戏的同时也加入了很多新元素。虽然有点简单，但是这款游戏成为后来 15 年里后复古类游戏的一个模板。

不过，尽管这个游戏也许是后复古美学的精神领跑者，但是也很难将所有的此类游戏跟这款游戏直接联系起来。因为从 1993 年这款游戏发布到 2007 年这类游戏大量出现之间，没有什么其他的此类游戏出现。

与 Tempest 2000 不一样的是，在接下来的几年里，复古游戏开发转向了不同的方向。当 Hasbro 在 1998 年收购了 Atari 的资产后，他开始创造一些 Atari 经典游戏的现代版，其中有 Missile Command，Pong，Breakout 和 Centipede。与此同时，Activision 针对 Battlezone 和 Asteroids

也做出同样的决策。虽然有一些是好游戏，但是他们不是真正的后复古类游戏，因为他们更像用现代图像技术复刻了老游戏。

复古游戏集合相同的地方是：复古游戏的套装软件在 ps2 上取得了很大的成功（例如：Activision Anthology，Capcom Classics，Namco Museum）。

虽然一些集合用一些微小的修改提供了这些游戏的混合和整理版本，但是大部分的是靠完全复制游戏机上经典的的游戏的游戏体验赚钱。

虽然其他的后复古游戏是肯定存在的（如果你有发给我们），但直到目前这一代的游戏机，我们才看见这类游戏慢慢成形。这了游戏真正

的出现，是在 Xbox Live Arcade，PlayStation Store 和 WiiWare 游戏机下载区出现后出现的，Web 版的 flash 游戏也如病毒般的爆发了。

这里有一些你十分熟悉的后复古类游戏的列子：

Geometry Wars：最好的之一，第一个后复古游戏的例子应该就是 Geometry Wars。第一次发布在 Xbox Live Arcade 是 2007 年，这个游戏是一剂猛药，证明销售好的游戏可以由一个非常小的小组创作出来并且能在 Xbox Live Arcade 上成功。这个有明显的复古外观，但是它加入了一个现代的操作机制，迷幻音乐和其他的效果。它是首个和最好的新后复古运动的例子之一。它相比榜上其他的游戏同样既是有创意的并且在收入上也是成功的。



Space Giraffe：自从我们赞颂 Jeff Minter 作为这个类别的创立者之一，添加他自己的一部游戏到这个列表应该很公平。Space Giraffe 是 Tempest 2000 的一个进化，和也许是它最大的败笔。在这个游戏包含了大部分的后复古美学（包括实际上来自一部 Minter 的原版电子游戏的声音），并且它也包含了一些 Tempest 模板的重要变更的同时，他的玩法非常接近于经典的复古游戏。尽管有这么多的继续，以至于他变成了最难玩的游戏而不是快速上手的游戏。这也许能解释相比其他在这个列表里的游戏，它有限的影响和低销售量。

Pac-Mac Championship Edition (CE)：Pac-Man CE 是一个最怪但是上瘾的怪兽，并且它是后复古气质最好的例子之一——这个经典游戏的再造。这个游戏拥有 Pac-Man 的精髓(吃掉所有的豆子，吃掉力量增强来杀死鬼魂)，并且最大限度的原本的保持了它。尽管如此，它添加变形迷宫和改进的玩法来使整个游戏和原始的不同，这就足以改变了娱乐性。你永远不能在 Pac-Man CE 里完成一个等级，你只能一个等级的一部分，然后游戏场地改变了他自己，而你则继续游戏。这个并非娱乐性核心的微小的改变是整个都不同并且上瘾的。玩这个游戏的时候你会觉得它和 Pac-Man 的后续版本，但是那个时候设计的游戏状态不能产生这样的游戏。只是一个很重要的差别，相比一些最好的后复古游戏：隐隐的感觉是一些这类游戏不能挽回电视游戏的黄金时期。

Bit Boy：Bit Boy 是很像复古游戏挑战赛，它将玩家带入不同的级别，每个灵感来自一个不同的时期的电视游戏图像和玩法。他不包含多人游戏，他将玩家带入了一个复古游戏的旅行，超过整个类别而创建了完全新式的东西。

后复古病毒 Flash 游戏：在这些商业化的游戏里，许多 Flash 病毒游戏也能算作后复古游戏类别。

## 后 复 古 游 戏 对 开 发 人 员 意 味 着 什 么

对于我们来说，后复古游戏的类型无疑存在并且一直成长。游戏开发人员能从中学习什么呢？

好的，首先，许多（但不是所有）这些游戏已经十分成功，这个事实不能被忽视。这说明购买可下载游戏和玩 flash 小游戏的人非常喜欢这个类型。这也意味着复古游戏的爱好者在变的更加成熟。你不能指望靠一个纯粹的复古风格游戏去发展任何的爱好者。更有可能是通过加入一些内容而不是组合原有的内容来让游戏获得关注。

如果你对制作这样的游戏感兴趣，我们有些建议：看一看这些老游戏，并且试着想象一下他们做成后复古游戏是什么样子。在游戏第一次发布的时候，那些你加入的经典概念不能完成。他能够超越或者解构原有游戏，还是上面堆积现代设计元素，都是还是都不是。我们是真正痴迷于这个新兴的趋势，我们期待着玩在这一流派的新游戏。

在这一章里，我们将会制作一个叫做 Blaster Mines 的游戏，它属于后复古游戏或者超复古类游戏的分支。

## 设 计 Blaster Mines

我们开始先看一看一张 Blaster Mines 游戏的屏幕截图。



Blaster Mines 将会是一个 Asteroids 或者 Geometry Wars 风格的游戏。基础的技术说明勾勒出了游戏的名字，灵感，目的和用一小段描述玩法。

游戏名：Blaster Mines

游戏类型：在 360 度可滚动世界里的长廊爆破手。

游戏灵感：Atari 的 Asteroids 是，并且永远是我们任何时候都特别喜欢的街机游戏之一。过去的四年里，有 10 款左右的 Asteroids 派生版本被开发出来。这种类型的游戏作为五或者十分钟时长的游戏十分合适，但是更重要的是，测试游戏优化的最好途径。当我们开始使用一个新的开发语言环境时，我们可能总是回到这种类型的游戏上，只是想知道我们能将这个游戏开发环境使用到什么样的极限。这种游戏就是要炸掉屏幕上无尽波数的进攻者，坚持得越长越好。Geometry Wars 的出现给这类游戏带来了新的控制方式，滚屏，威力增强和更多其他内容。这类游戏看起来难以正确的编写。最难得是，当开发者想要添加上百个导弹，粒子，和敌人到游戏屏幕上的时候。这样导致 Flash 引擎变慢。

游戏的目标：你是一个孤独的海军陆战队员，尝试穿过一个有无数波敌人，可向任意方向滚屏的雷场，来找到回家的路。

游戏玩法：玩家必须开着飞船，通过一个 360 度滚屏的世界，并且炸掉美关所有的地雷才能到下一关。

升级：等级的提升经过简单的计算会增加地雷的数量和速度。一共有 10 关。每关背景和地雷的颜色都会变化。下面是一些基础的技术设计的规格：

世界大小:800\*800(非 tiles)

摄像机大小:400\*400

滚屏类型:基于 BitmapData 容器

用户操作：玩家将跟随鼠标并自动开火。



资源格式:将向量缓存在了一个 BitmapData 的数组里 ( 或者一个 BitmapData 实例 )。

## 这章里的游戏开发概念

在这章里我将覆盖到下面这些内容：

垃圾回收管理

优化渲染速度

优化内存使用

用渲染外形创建一个基于时间的步进计时器

给框架加一个帧率计时器

在代码里绘制矢量资源

用 BitmapData 数组缓冲矢量资源，以便渲染

使用 ScrollRect 优化在 BitmapData 容器上的基于屏幕的滚屏

使用 BitmapData 来创建一个简单的雷达屏幕

使用外观表格

创建一个简单的粒子引擎

缓冲对象

## 修改游戏框架

我们将要把如下功能添加到我们不断改进的游戏框架里：

静音功能: 这个将由主程序控制，触发一个 SoundManage 类里的新函数。

暂停功能:这个将在 GameFrameWork.as 里实现，仅能那些使用在本章里创建的这种基于时间的步进计时器的游戏里可用。我们这样做是为了让 GameFramework 文件能够兼容前面用标准计时器创建的所有游戏。

基于时间的步进计时器 用一个基于 ENTER\_FRAME 的步进计时器来替代基于 Timer 的游戏计时器。这种类型的计时器是为了保证当在帧率变化的情况下，游戏渲染和物体移动保持不变。

帧率的分析器：在玩游戏之前用这个来测试玩家机器性能。结果会传给 Game 类，基于测试出来的帧率来增加或者降低游戏效果。

帧计数器：帧计数器是用来计算当前的工作帧率。帧率分析器会用到它，也能在游戏的过程中显示出来。注意：在本章末尾有

整个 GameFrameWork.as 类代码。建议你在了解所有改变的内容在使用这个版本。

测试舞台是否可访问



GameFrameWork.as 文件第一个修改是添加代码，在舞台可用的时候 Main 能够调用它。这样做是因为 GameFrameWork.as 会需要为暂停和静音键在舞台上添加监听，而在这两个功能触发的时候舞台必须是可用的。

## 创建新的 addedToStage 函数

将这个新函数放在 GameFrameWork.as 文件里

```
public function addedToStage(e:Event = null):void {
    stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
    this.focusRect = false;
    stage.focus = stage;
    trace("gamFrameWork added to stage");
}
```

这个函数主要的目的是，仅在舞台对 SWF 文件可用后添加一个舞台的引用。这样可以保证 Main 的 init() 函数在舞台可用之前不会被调用。

在这个章节的“添加暂停和静音到框架里”小结里，我们将会检查这个函数是如何被调用的。主函数将会复写这个函数，先调用自身这个函数再通过 super.addToStage() 调用父类的这个函数。

## 添加暂停和静音功能到框架里

通过在 GameFrameWork.as 里添加一系列的新的键监听器来将静音功能添加进来。当 M 键被按下的时候，这些监听器会监测到然后抵用一个 SoundManager 里的函数。P 键的暂停功能用的是同一波监听器来触发。

Game 类负责游戏的具体暂停处理。为了实现游戏的暂停功能，我们会复写 Main.as 类里的 systemGamePlay 函数并且当我们调用 runGame 函数会传个暂停的类变量给 Game 类。这些最后实际上是有新的步进计时器的 runGameTimeBased 函数完成的。

为了保证这本书里之前章节创建的游戏仍然能和这个框架一起编译，我们只会将暂停功能放入 runGameTimeBased 函数，这个是为基于时间的步进计时器而创建的，而不是将它添加到之前所有游戏使用到的 runGame 函数。

这里有一些暂停和静音功能使用的 public 变量：

```
public static const KEY_MUTE:int = 77; // added chapter 11
public static const KEY_PAUSE:int = 80; //added chapter 11
public var paused:Boolean = false;
public var pausedScreen:BasicScreen;
```

现在，让我们仔细的看看处理暂停的静音功能的五个新函数。

private function keyDownListener(e:KeyboardEvent):void:用一个 switch :case 语句来决定判定来判定在某个键被按下是否需要做些操作。M 键触发静音，P 键触发暂停，具体的处理函数将在后面提到。



`private function pausedKeyPressedHandler():void` : P 键按下的时候调用这个函数，将新的布尔类变量 `paused` 设置为 `true`，同时显示 `pausedScreen`。

`PausedScreen` 是一个 `BasicScreen` 的一个简单实例，所以我们能监听到屏幕上 Ok 按钮的点击事件。这是游戏玩家想要关闭 `pausedScreen` 的信号。这个函数将这个监听添加到 `pausedScreen` 的 Ok 按钮的点击事件上：`pausedScreen.addEventListener(CustomEventButtonId.BUTTON_ID, pausedScreenClickListener, false, 0, true);`

这个函数同时也将 `pausedScreen` 添加到显示列表。

`public function pausedScreenClickListener(e:Event):void`:这个函数将会把 `pausedScreen` 从显示列表移除，移除 `pausedScreen` 上 Ok 按钮的事件监听并且将类变量 `paused` 设置为 `false`。

`private function muteKeyPressedHandler():void`:当 M 键按下时，这个函数会调用 `soundManager.muteSound` 函数。在这个小节找到这个函数的描述 on changes to the `SoundManager` class.

还需要添加其他的一些常量到 `com.efg.framework.FrameWorkStates.as` 文件里：`public static const STATE_SYSTEM_PAUSE:int = 99;`

Main 类的完整代码参见“The Main Class”章节。`GameFrameWork` 类的完整代码参见“The full GameFramework class code”章节。

## 增加基于时间的步进计时器

在第二章，我们介绍了基本的计时器，用于我们游戏里的循环。这个计时器通过一个 `Timer` 类的实例实现，运行在一个基于 `frameRate` 变量的中断上。如果我们把 `frameRate` 变量设置为 40,计时器将会每隔  $1,000/40$  微妙运行一次。在这样一个计时器的设置下

，能使当前的这些游戏运行得不错。但是如果游戏在各种机器和 Flash Player 版本下运行，需要注意帧率的差别。这个差别会导致有些的 `sprites` 比我们期望的移动得快或者慢。

使用计时器时，我们把游戏的运行等分成每个游戏循环迭代用的一小片，并且希望游戏能在相等的时间片里处理所有的代码和渲染屏幕。所以当我们在 `Drive She Said` 中以每次 4 个像素的速度移动小汽车，我们只是希望这个小汽车每次移动 4 个像素，并且希望在一秒中内个小汽车能移动  $30*4$ （按照我们的帧率计时的速度值，单位是像素/每秒）。因为 `Drive She Said` 真的不是一个大计算量的游戏，我们能达到我们期望的每秒 120 像素移动速率。虽然不是经常能这样。如你所见，在 `Blaster Mines` 里我们将会在游戏里创建很多移动的对象，以至于系统和平台也许能很好的处理。因此，我们在每个帧运行的时间片上增加一些功能，来保证每秒钟所有的对象能按照我们期望的距离移动。结果是，在帧计时下，对象不总是移动完全同样的距离，但是每秒钟，我们将能保证对象移动的距离是我们想要的。所以，计时 Flash Player 在一个低于我们期望的帧率下运行，我们也能保证游戏里的物体能按照我们想要的结果移动。

在这章里，我们将会创建一个新型的计时器叫做基于时间的步进计时。这个计时器将会用 `ENTER_FRAME` 时间实现，而不是 `Timer` 类。我们将会用舞台的帧率替代依赖于 `Timer` 间歇调用，来运行



我们的游戏。虽然我们不是简单的依赖于舞台的帧率来保持我们的游戏的流畅运行。我们分析每帧运行的时间，并且根据这个时间修改该我们游戏角色移动的像素数。

首先，我们需要去在 startTimer 函数里添加使用新计时器的能力。

```
public function startTimer(timeBasedAnimation:Boolean=false):void {
    stage.frameRate = frameRate;
    if (timeBasedAnimation) {
        lastTime = getTimer();
        addEventListener(Event.ENTER_FRAME, runGameEnterFrame);
    }else{
        timerPeriod = 1000 / frameRate;
        gameTimer=new Timer(timerPeriod); /*** changed removed in new chapter 2
        gameTimer.addEventListener(TimerEvent.TIMER, runGame);
        gameTimer.start();
    }
}
```

startTimer 函数现在接受布尔参数，这就允许我们在原来的计时器和新的基于时间的步进计时器之间切换。参数为 true 时候我们就进入了 ENTER\_FRAME 版本。这个时候我们的游戏循环将会在每帧运行的时候调用 runGameEnterFrame 函数而不是原来的 runGame 函数。

## 添加 runEnterFrame 函数

runEnterFrame 的函数用一个 getTimer 调用来计算运行时每帧之间的为微秒数。这是一个被存储在 timeDifference 里的类级别变量。

```
public function runGameEnterFrame(e:Event):void {
    timeDifference = getTimer() - lastTime
    lastTime = getTimer();
    systemFunction();
    frameCounter.countFrames();
}
```

每个游戏的 Main.as 类将会复写 GameFrameWork 的 systemGamePlay 状态函数，并且将 timeDifference 和暂停变量（如果设置为 true 就会使游戏暂停）一起传入 BlasterMines 的 runGameTimeBased 函数。这是一个我们将添加到 Game 基类的一个新函数。下面是这个这个函数的例子，属于 BlasterMines 创建的 Main.as 类。

```
override public function systemGamePlay():void {
    game.runGameTimeBased(paused,timeDifference);
}
```



```
}
```

我们将在 Game 基类里创建这个基于 runGameTime 的函数，并且将在每个需要实现这个计时器的游戏里复写它。BlasterMines 里的 runGameTimeBase 函数将跟下面的一样。

```
override public function  
runGameTimeBased(paused:Boolean=false,timeDifference:Number=1):void {  
if (!paused) {  
systemFunction(timeDifference);  
}  
}
```

这个函数是 BlasterMines.as 内置状态机的一部分。systemFunction 将会持有当前状态函数的引用。这些函数必须接受 paused 和 timeDifference 变量。

这里是一个 BlasterMines.as 的 systemGamePlay 函数的例子。注意，暂停功能在之前的 runGameTimeBased 函数里就实现了，所以如果 paused 为真，systemGamePlay 也不会被调用。

```
private function systemGamePlay(timeDifference:Number=0):void {  
update(timeDifference);  
checkCollisions();  
render();  
updateScoreBoard();  
checkforEndLevel();  
checkforEndGame();  
}
```

这个为 Blaster Mines 升级的函数必须接受 timeDifference 值。这里有一小段这种函数的示范：

```
private function update(timeDifference:Number = 0):void {  
//time based movement modifier calculation  
var step:Number = (timeDifference / 1000)*timeBasedUpdateModifier;  
trace("timeDifference= " + timeDifference);  
trace("timeDifference/1000=" + String (timeDifference / 1000));  
trace("timeBasedUpdateModifier=" + timeBasedUpdateModifier);  
trace("step=" + step);  
...  
}
```

真正的处理在这里。timeDifference 传进了更新函数，然后我们计算处理这个值会对每个游戏角色的移动怎样造成影响。我们用一个叫 timeBasedUpdateModifier 的变量来计算，我们在 Game.as 基类将要创建的这个值，这个变量就是我们想要游戏每秒刷新的次数。这个跟游戏的帧率是一样的。这个值会在基于舞台帧率的基础上，在 Main 类里赋值。

我们计算当前这一步的是用 timeDifference 除以 1,000（一秒钟里的为微秒数）然后在乘以 timeBasedUpdateModifier，这个就是每个游戏角色移动的脚步的修正值。



例如，如果 timeDifference 是 20,意味着上一帧运行时间是 20 微秒，我们例子中的帧率是 40 ( timeBaseUpdateModifier=40 ) ,所以一步大概是：

```
timeDifference= 20
timeDifference/1000=0.02
timeBasedUpdateModifier=40
step=0.8
```

step 就是我们每个游戏的角色在在一帧中移动距离的百分比。每个游戏角色计算移动距离要乘以这个值。这里有一个 BlasterMines 里计算 Mine 敌人角色移动距离的计算，这一部分就在我们将来创建的游戏的 Mine 类里。

```
public function update(step:Number=1):void {
//trace("updateModifier=" + updateModifier);
nextX+=dx*speed*step;
nextY+=dy*speed*step;
```

就像你在代码里看到的，和 step 相乘之后得到了 nextX 和 nextY 值。这个计算让我们能基于当前帧率修改移动距离的百分比。所有这些函数完整的代码，在我们创建了完整 BlasterMines 游戏后呈现出来。

## 优化渲染分析器

我们创建的 FrameRateProfiler 类是用来分析用户系统是否能在我们期望帧率运行游戏。

这类将是我们 Blaster Mines 游戏创建的一个新文件。他将是游戏框架的一部分。

这里是类文件名和位置

/source/classes/com/efg/framework/FrameRateProfiler.as

这里是 FramRateProfiler 类的完整代码。在阅读完整段代码后，我们会来讨论细节。

```
package com.efg.framework
{
import flash.display.*;
import flash.events.*;
import flash.utils.Timer;
import flash.geom.*;
import flash.events.*;
import flash.utils.getTimer;
import flash.text.*;
import flash.events.EventDispatcher;
/**
 * ...
 * @author Jeff Fulton
 */
```



```
public class FrameRateProfiler extends Sprite {
    public static const EVENT_COMPLETE:String = "profile complete";
    private var profilerTimer:Timer;
    //public preperities
    public var profilerRenderObjects:int = 500;
    public var profilerRenderLoops:int = 10;
    public var profilerDisplayOnScreen:Boolean = false;
    public var profilerXLocation:int = 0;
    public var profilerYLocation:int = 0;
    public var profilerFrameRateAverage:int = 0;
    private var profilerFrameRate:int = 40;
    private var profilerRenderFrames:int = 0;
    private var profilerBackground:BitmapData = new BitmapData(400, 400, false, 0x0000000);
    private var profilerCanvas:BitmapData = new BitmapData(400, 400, false, 0x0000000);
    private var profilerBitmap:Bitmap = new Bitmap(profilerCanvas);
    private var profilerObject:BitmapData = new BitmapData(20, 20,false, 0xff0000);
    private var profilerFrameRateTotal:int=0;
    private var profilerFrameRateEventCounter:int = 0;
    private var profilerFrameCount:int = 0;
    private var profilerTempObject:Object;
    private var profilerObjectArray:Array = [];
    private var profilerRenderPoint:Point = new Point(0, 0);
    private var profilerFrameRateArray:Array= [];
    private var format:TextFormat=new TextFormat();
    private var messageTextField:TextField = new TextField();
    private var frameCounter:FrameCounter = new FrameCounter(); //added chapter 11
    public function FrameRateProfiler() {
        addChild(frameCounter);
        frameCounter.x = 0;
        frameCounter.y = 0;
    }
    public function startProfile(frameRate:int):void {
        trace("start profile");
        profilerFrameRate = frameRate;
        if (profilerDisplayOnScreen) {
            profilerBitmap.y = 20;
            profilerBitmap.x = 0;
        }else {
            profilerBitmap.x = stage.width + 10;
        }
    }
}
```



```
addChild(profilerBitmap);
format.align = "center";
format.size=24;
format.font="Arial";
format.color = 0xffffffff
format.bold = true;
messageTextField.defaultTextFormat = format;
messageTextField.text = "Profiling\nOptimal\nFrame Rate";
messageTextField.width=200;
messageTextField.height = 200;
messageTextField.x = 100;
messageTextField.y = 100;
addChild(messageTextField);
for (var ctr:int = 0; ctr < profilerRenderObjects; ctr++) {
    profileAddObject()
}
profilerRenderFrames = profilerRenderLoops * profilerFrameRate;
addEventListener(Event.ENTER_FRAME, runProfile);
}
private function runProfile(e:Event):void {
    profileUpdate();
    profileRender();
    if (frameCounter.countFrames()) {
        profilerFrameRateTotal += frameCounter.lastframecount;
        profilerFrameRateEventCounter++;
        messageTextField.text = "Profiling\nOptimal\nFrame Rate\n" + _
String(int(profilerFrameCount / profilerRenderFrames * 100)) + "%";
        profilerFrameRateArray.push(frameCounter.lastframecount);
    }
    profilerFrameCount++;
    if (profilerFrameCount > profilerRenderFrames) {
        profileCalculate();
    }
}
private function profileCalculate():void {
    profilerFrameRateAverage = profilerFrameRateTotal / profilerFrameRateEventCounter;
    dispose();
    dispatchEvent(new Event(EVENT_COMPLETE));
}
private function profileAddObject():void {
```



```
var profilerTempObject:Object = new Object();
profilerTempObject.x=(Math.random() * 399);
profilerTempObject.y=(Math.random() * 399);
profilerTempObject.speed = (Math.random() * 5) + 1;
profilerTempObject.dx=Math.cos(2.0*Math.PI*((Math.random()*360)-90)/360.0);
profilerTempObject.dy = Math.sin(2.0 * Math.PI * ((Math.random()*360) - 90) _
/ 360.0);
profilerObjectArray.push(profilerTempObject);
}
private function profileUpdate():void {
for each (profilerTempObject in profilerObjectArray) {
profilerTempObject.x += profilerTempObject.dx * profilerTempObject.speed;
profilerTempObject.y += profilerTempObject.dy * profilerTempObject.speed;
if (profilerTempObject.x > profilerCanvas.width) {
profilerTempObject.x = 0;
}else if (profilerTempObject.x < 0) {
profilerTempObject.x = profilerCanvas.width;
}
if (profilerTempObject.y > profilerCanvas.height) {
profilerTempObject.y = 0;
}else if (profilerTempObject.y < 0) {
profilerTempObject.y = profilerCanvas.height;
}
}
}
private function profileRender():void {
profilerCanvas.lock();
profilerRenderPoint.x = 0;
profilerRenderPoint.y = 0;
profilerCanvas.copyPixels(profilerBackground, profilerBackground.rect, _
profilerRenderPoint);
for each (profilerTempObject in profilerObjectArray) {
profilerRenderPoint.x = profilerTempObject.x;
profilerRenderPoint.y = profilerTempObject.y;
profilerCanvas.copyPixels(profilerObject, profilerObject.rect, _
profilerRenderPoint);
}
profilerCanvas.unlock();
}
public function dispose():void {
```



```
removeEventListener(Event.ENTER_FRAME, runProfile);
for (var ctr:int = 0; ctr < profilerObjectArray.length; ctr++) {
    profilerObjectArray[ctr] = null;
    profilerObjectArray.splice(1, 0);
}
removeChild(profilerBitmap);
removeChild(messageTextField);
profilerObjectArray = null;
profilerBackground.dispose();
profilerBackground = null;
profilerCanvas.dispose();
profilerBitmap = null;
profileTimer = null;
format = null;
messageTextField = null;
frameCounter = null;
}
}
}
```

## 设计帧率分析器技术设计

让我们先来看一看需要为分析器定义函数和变量。这里是公共变量：

```
_ public var profilerRenderObjects:int = 500;; 分析器里使用的对象个数
_ public var profilerRenderLoops:int = 10;; 运行分析器的迭代次数
_ public var profilerDisplayOnScreen:Boolean = false;; 屏幕上分析器显示的开关。
_ public var profilerXLocation:int = 0;设置分析器左上角 x 位置
_ public var profilerYLocation:int = 0;设置分析器左上角 y 位置
```

这个和前面 4 个变量在游戏的主文件 Main.as 文件里，他们在分析器开始之前能够设置成一些制定的值。这些值可以根据每个你创建的游戏不同而自定义。

```
_ public var profilerFrameRateAverage:int = 0;;当分析完成的时候保存结果，就是，在分析期间用户机器的平均帧率
```

这些是私有变量：

```
_ private var profilerFrameRate:int = 40;;游戏的设计帧率
```



\_ private var profilerRenderFrames:int = 0;;分析器运行的帧数，基于传入的 profilerRenderLoops 计算得出

\_ private var profilerBackground:BitmapData = new BitmapData(400, 400, false, 0x000000);分析器屏幕的背景

\_ private var profilerCanvas:BitmapData = new BitmapData(400, 400, false, 0x000000);显示分析对象的 BitmapData 容器

\_ private var profilerBitmap:Bitmap = new Bitmap(profilerCanvas);显示分析结果的显示对象

\_ private var profilerObject:BitmapData = new BitmapData(20, 20,false, 0xff0000);分析结果的外观

\_ private var profilerFrameRateTotal:int=0;分析 session 里收集的帧率之和

\_ private var profilerFrameRateEventCounter:int = 0;分析事件之和（每秒钟触发一个事件）

\_ private var profilerFrameCount:int = 0;分析器运行的总帧数

\_ private var profilerTempObject:Object;分析渲染阶段使用的临时渲染对象。

\_ private var profilerObjectArray:Array = [];一个渲染的分析对象数组

\_ private var profilerRenderPoint:Point = new Point(0, 0);

用来将渲染对象 blit 到屏幕上的公共点。（blit：其意义是将一个平面的一部分或全部图象整块从这个平面复制到另一个平面。）

\_ private var profilerFrameRateArray:Array = [];每秒钟加 1 的帧率数组

\_ private var format:TextFormat = new TextFormat();显示文字的格式

\_ private var messageTextField:TextField = new TextField();渲染百分比文字显示的文字框

\_ private var frameCounter:FrameCounter = new FrameCounter();一个 FrameCounter 的实例（一个新自定义类）

下面是公共函数

\_ public function FrameRateProfiler():

构造还是没有参数。他添加了一个 FrameCounter 的实例，这个我们将在下一节创建，并且还把他放在了屏幕里。

addChild(frameCounter);

frameCounter.x = 0;

frameCounter.y = 0;



```
_ public function startProfile(frameRate:int):void:
```

这个函数需要 frameRate 参数。分析器将使用这个值来估计用户的机器运行帧率的能力。这个函数设置了分析的文字和位置，在分析器开始之前。这个函数有五个十分重要的地方。

```
for (var ctr:int = 0; ctr < profilerRenderObjects; ctr++) {  
    profileAddObject()  
}
```

这里我们将创建那些基于 profilerRenderObjects 的值，真正会被渲染在屏幕上的对象。

```
_ profilerRenderFrames = profilerRenderLoops * profilerFrameRate;
```

这里，我们将计算我们希望分析器运行的帧数。这基于我们希望分析器迭代的次数和我们希望的运行帧率。

```
_ addEventListener(Event.ENTER_FRAME, runProfile);
```

分析器由 ENTER\_FRAME 事件触发，并且将会在每次迭代的时候调用 runProfile 函数。它将会运行 profilerRenderLoops 次迭代。

下面是私有函数：

```
_ private function runProfile(e:Event):
```

这个函数分析的游戏循环。它的工作就是为每个渲染循环计算帧率，并且在分析完成后将数据储存起来用以分析。

```
if (frameCounter.countFrames()) {  
    profilerFrameRateTotal += frameCounter.lastframecount;  
    profilerFrameRateEventCounter++;  
    messageTextField.text = "Profiling\nOptimal\nFrame Rate\n" + _  
String(int(profilerFrameCount / profilerRenderFrames * 100)) + "%";  
    profilerFrameRateArray.push(frameCounter.lastframecount);  
}
```

当 if 条件里的 frameCounter 返回一个真的时候，1000 微秒过去，距离上一个帧率事件。一个帧率事件是一个在 1000 微秒后计算当前帧率的调用。它不是一个实际的事件实例但是比我们自己临时构想出来的术语要好。因为我们不知道到底系统会怎样在分析器下运行，所以我们不能假定测试所设定的循环次数与实际相符。我们也不知道在压力测试下系统将会是什么表现。因为如，我们使用 profilerFrameRateEventCounter 来保存我们事件的计数。我们将 frameCounter 的最后计算的帧率加起来，profilerFrameRateTotal 并且更新屏幕里分析的文字。每个帧率也放在 profilerFrameRateArray 数组里，用来计算平均值。

```
_ private function profileCalculate():void:
```



一旦分析完成了，我们必须计算平均的帧率，来保证使用真实的帧率事件。然后我们调用析构函数并且发送一个主程序监听的事件，这样就能跳到下一个框架状态了。

```
profilerFrameRateAverage = profilerFrameRateTotal / profilerFrameRateEventCounter;
```

```
dispose();
```

```
dispatchEvent(new Event(EVENT_COMPLETE));
```

```
_ private function profileAddObject():void:
```

这个函数为分析过程添加对象。仅供分析器系统内部使用。

```
_ private function profileUpdate():void:
```

这是分析系统内部使用的函数，这个是跟新分析器对象位置的。

```
_ private function profileRender():void:
```

分析器运行时，将 blits 所有对象到 canvas 里。

```
_ public function dispose():void:
```

摧毁分析器中使用对象来释放内存。

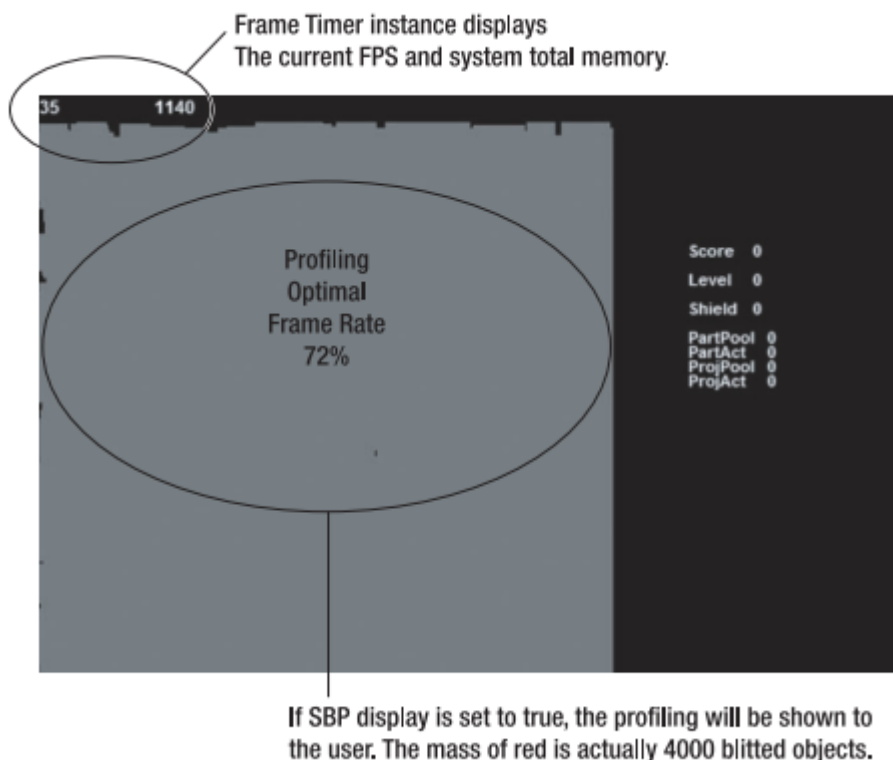


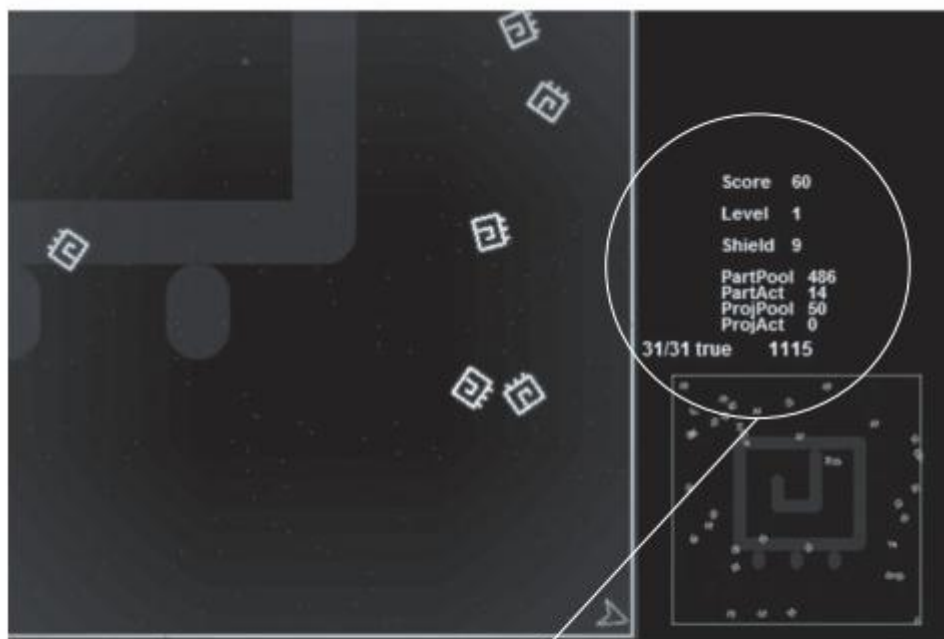
Figure 11-2The render profiler in action

## 监视帧率和内存使用



有两个相关的简单工具我们能用来检查游戏在 Flash 里是否正常运行。第一个就是观察系统当前的运行帧率,第二个就是观察系统运行的总内存量的变化。

本书中你已经了解和在本章中的优化方案会影响到这些值。我们现在创建一个简单的类,你能够放在屏幕上(如果你喜欢的话)显示这两个重要计量的当前状态。



Under the ScoreBoard we have placed the FrameTimer instance. The current frame rate is 31 the desired frame rate is 31. The true means that updateAfterEvent is on. 1115 pages of 4096 memory is being consumed.

FrameCounter 类是 FrameRateProfiler 用来执行分析器的帧率事件的。一个帧率事件每 1000 微秒触发一次。在游戏运行的时候,我们也能将 FrameCounter 添加到屏幕上,它显示了分析出来的帧率,当前帧率,当前系统使用的内存。

这个类将作为我们为 Blaster Mines 游戏创建的第二个类。他将是游戏框架包结构的一部分。

下面是这个类的文件名和位置。

/source/classes/com/efg/framework/FrameCounter.as

FrameCounter 类也显示了 Flash Player 和游戏使用的当前系统内存。

这里是 FrameCounter 类的完整代码。在完整阅读完这段代码后我们会详细的讨论它。

```
/**
 * ...
 * @author Jeff Fulton
 * @version 0.1
 */
```



```
package com.efg.framework {
import flash.display.*;
import flash.events.*;
import flash.system.System;
import flash.utils.getTimer;
import flash.text.TextField;
import flash.text.TextFormat;
public class FrameCounter extends Sprite{
private var format:TextFormat=new TextFormat();
private var framectrText:String;
private var textColor:uint = 0xffffffff;
private var memoryUsedText:String;
private var framectrTextField:TextField = new TextField();
private var memoryUsed:TextField = new TextField();
public var lastframecount:int = 0;
private var frameLast:int = getTimer();
private var frameCtr:int = 0;
public var showProfiledRate:Boolean = false;
public var profiledRate:int;
public function FrameCounter():void {
format.size=12;
format.font="Arial";
format.color = String(textColor);
format.bold = true;
framectrText="0";
framectrTextField.text=framectrText;
framectrTextField.defaultTextFormat = format;
framectrTextField.width=80;
framectrTextField.height = 20;
addChild(framectrTextField);
memoryUsedText = "0";
memoryUsed.text=memoryUsedText;
memoryUsed.defaultTextFormat = format;
memoryUsed.width=100;
memoryUsed.height = 20;
memoryUsed.x = 80;
addChild(memoryUsed);
}
public function setTextColor(color:uint):void {
format.color = String(color);
```



```
}  
public function countFrames():Boolean {  
    frameCtr++;  
    if (getTimer() >= frameLast + 1000) {  
        lastframecount = frameCtr;  
        if (showProfiledRate) {  
            framectrText = frameCtr.toString() + "/" + profiledRate;  
        }else{  
            framectrText = frameCtr.toString();  
        }  
        framectrTextField.text = framectrText;  
        frameCtr = 0;  
        frameLast = getTimer();  
        memoryUsedText = String(System.totalMemory / 1024);  
        trace(memoryUsedText);  
        memoryUsed.text=memoryUsedText+"kb";  
        return(true);  
    }else {  
        return(false);  
    }  
}  
} // end class  
} // end package
```

我们先看一看 FrameCounter 类的技术定义，从公共变量开始。

```
_ public var lastframecount:int:
```

这个变量保存了最近 100 微秒内的帧数。

```
_ public var showProfiledRate:Boolean:
```

这个布尔变量从 Main 外部设定，决定帧率的显示格式，如果为真就显示为 当前/分析后的，如果为假就显示为当前帧率。

```
_ public var profiledRate:int:
```

这个变量保存了分析过的帧率，以备显示需要。

接下来是私有变量

```
_ private var format:TextFormat:
```

文本显示格式

```
_ private var framectrText:String:
```



表示当前帧数的字符串

```
private var memoryUsedText:String:
```

表示当前内存使用值的字符串

```
_ private var framectrTextField:TextField:显示 framectrText 字符串的地方
```

```
_ private var memorypagesField:TextField: 显示 memorypagesText 的地方
```

```
_ private var frameLast:int:包含上次计数时间的微秒数
```

```
_ private var frameCtr:int: 包含了 1000 毫秒事件里的帧数
```

下面是公共函数

```
_ public function FrameTimer():void:不带参数的构造函数.它的功能是设定显示帧率计数器和内存使用指示器的文本区域。
```

```
_ public function countFrames():Boolean:
```

这个函数不带任何参数,但是如果从上一帧时间过去有 1 秒钟了就返回真。为了让 FrameCounter 运行正常,在 Main 或者 FrameRateProfiler ( 或者再任何你创建来监视 FPS 的 ) 里的每个游戏计时器的最后调用 countFrames。

整个函数的内容被一个条件判断包含着,这个条件判断着是否从上次帧计数时间开始已经过去了 100 微秒 ( 1 秒 )。

```
if (getTimer() >= frameLast + 1000) {  
... do all of the frame count event code  
return(true);  
}else {  
frameCtr++;  
return(false);  
}
```

如果从最后一次帧计数事件 100 微秒没有过去,我们就在 framCtr 变量上加 1。

有帧计数事件的时候发生了什么呢?在 ActionScript 上下文里这个不是一个真实的事件,只是我们对每次一秒钟过去的时间的一个命名。

在这个事件里,我们想要显示在上一秒里记录的帧数,并且把这个信息放在 lastframecount 里,提供给本类以外的代码使用

我们也想去显示当前被使用的内存数并且重置为下一帧计数事件重置计数器。

```
if (showProfiledRate) {  
framectrText = frameCtr.toString() + "/" + profiledRate;
```



```
}else{
    framectrText = frameCtr.toString();
}
framectrTextField.text = framectrText;
frameCtr = 0;
frameLast = getTimer();
memoryUsedText = String(System.totalMemory / 1024);
trace(memoryUsedText);
memoryUsed.text=memoryUsedText+"kb";
return(true);
```

这些就是这个类功能的全部内容。现在，让我们继续看看 com.efg.framework.Game.as 里需要改变的地方。

## 修改 Game 类

Game.as 类有需要修改几个地方来实现我们在这章里要添加的功能。现在，让我们迅速的看一下这个类。下面就是完整的 com.efg.framework.Game.as 类。

```
package com.efg.framework
{
    // Import necessary classes from the flash libraries
    import flash.display.MovieClip;
    import com.efg.framework.CustomEventScoreBoardUpdate;
    import com.efg.framework.CustomEventLevelScreenUpdate;
    /**
     * ...
     * @author Jeff Fulton and Jeff Fulton
     */
    public class Game extends MovieClip {
        //Create constants for simple custom events
        public static const GAME_OVER:String = "game over";
        public static const NEW_LEVEL:String = "new level";
        public var timeBasedUpdateModifier:Number = 40;
        public var frameRateMultiplier:Number = 1;
        //Constructor calls init() only
        public function Game() {}
        public function setRendering(profiledRate:int, framerate:int):void {}
        public function newGame():void {}
        public function newLevel():void {}
        public function runGame():void {}
        public function runGameTimeBased(paused:Boolean=false,timeDifference:Number=1):void {}
```



```
}  
}
```

我们添加了这些公共变量:

```
public var timeBasedUpdateModifier:Number = 40;  
public var frameRateMultiplier:Number = 1;
```

timeBasedUpdateModifier 已经在“添加基于时间的步进计时器”这一章里讨论过了。这个就是我们希望游戏运行的帧数。

添加 frameRateMultiplier 变量来,来允许游戏开发者去使用分析出来的帧率。例如,在 Blaster Mines, 如果分析出来帧率是设计帧率的 85%, 然后 multiplier 设置为 2。这样就让爆炸中的粒子翻倍了。

这里有新的公开函数:

```
public function setRendering(profiledRate:int, framerate:int):void {}  
public function runGameTimeBased(paused:Boolean=false,timeDifference:Number=1):void {}
```

setRendering 函数是用来设置 frameRateMultiplier 变量。profileRate 和 frameRate 两个作为参数传入。复写这个函数,开发人员可以自由的使用这些信息来设置游戏质量。当我们仔细的看一下游戏代码就会看到 Blaster Mines 包含一个很好的这样的例子。runGameTimeBased 函数在“添加基于时间的步进计时器”这一章里我们就已经讨论过了。

GameFrameWork 的 runGameEnterFrame 函数的 timeDifference 和 paused 值一起作为参数传入。

## 开始 Blaster Mines 项目

和这本书里所有的游戏一样,BlasterMines使用我们在第二章里创建的框架包结构。让我们在 FlashIDE 和 FlashDeveop(为了使用 Flex SDK) 里创建我们游戏需要的

## 在 FlashIDE 中创建 Blaster Mines 游戏项目

这一步需要再 FlashIDE 中创建。

1. 搭建你自己的 Flash 环境。我们用的是 CS3, 但是在 CS4 和 CS5 里这些也一样。
2. 在/source/projects/blastermines/flashIDE/folder 文件夹里创建一个叫 blastermines fla 文件
3. 在/source/projects/blastermines/flashIDE/folder 文件夹里, 为你的游戏创建一个叫 /com/efg/games/blastermines 的包结构
4. 设置 Flash 动画的帧率为 40PPS。设置宽为 600, 高为 400。
5. 设置文档类为 com.efg.games.blastermines.Main
6. 我们还没有创建 Main.as 类, 所以你会看见一个警告, 我们将会在这个章节的最后创建它。
7. 接下来, 把框架的可重用类包添加到 fla 文件的类路径。首先, 在 Publishing 设置里, 选择 Flash->ActionScript3 在 ActionScript Version 的下拉菜单里, 点击 Setting 按钮。



8. 然后，点击 Browse to Path 按钮，找到那个我们在第二章为包结构创建 source 文件夹。

9. 最后，选择类文件夹，点击 Choose 按钮，现在，当我们开始创建我们的游戏时就能使用 com.efg 框架

这里有 Flash IDE 版本的文件夹结构

```
[source]
[projects]
[blastermines]
[flexIDE]
[com]
[efg]
[games]
[blastermines]
```

## 在 Flash Develop 里创建 Blaster Mines 游戏项目

按照下面几步在 Flash Develop 里创建 Blaster Mines 游戏

1. 在/source/projects/blastermines 文件夹里创建一个叫 flexSDK 的文件夹( 如果你还没有这么做 )

2. 打开 Flash Develop，创建一个新的 Flex3 项目，取名 blastermines.位置必须在 /source/projects/blastermines/flexSDK 文件夹，包必须是 com.efg.games.blastermines

不要让 FlashDevelop 创建自动创建项目文件夹。确保 Create Folder For Project 是为选中的。

点击 Ok 按钮，创建项目。

3. 选择 Project>Properties>Classpaths 菜单选项并且点击 Add Class Path，按钮将框架的路径添加到项目里。

4.下一步，找到我们之前创建的 source 目录，选中包的子目录。

5.点击 Ok 按钮，然后是 Apply 按钮。

6. 选中 Project>Properties>Classpaths 项，改变输出的大小和帧率。设置帧率为 40,宽为 600，高为 400。

下面是 Flex SDK 版本的文件夹结构 ( 在 FlashDevelop 里的 )。

```
[source]
[projects]
[blastermines]
[flexSDK]
[bin]
[obj]
[lib]
[src]
```



[com]  
[efg]  
[games]  
[blastermines]

我们有了在框架之下创建游戏基本的结构。所以我们将讨论一个新主题，内容是关于框架类的结构，然后讨论创建可复用的框架代码。

在 Flex Builder FlashBuilder 或者其他的 IDE 中，请引用有关创建新项目和设置默认编译类的帮助文档。

如前所述，一个 Flash 开发的通用的方法，使用 FlashIDE 作处理和组织资源，使用 FlashDevelop 进行代码编辑。

如果你的工作流选择是这样的，你会希望照着 FlashIDE 的文件夹及包结构，而不是 FlexSDK 文件结构。

## 创建 Blaster Mines 的 Main.as 类

每个游戏都很不一样，所以我们必须为 Blaster Mines 创建一个独特的 Main.as 类，它将是基于第十章的 Main.as，但是在游戏和框架里，有很多针对 Blaster Mines 的变化。

Blaster Mines，接下来要创建的第二个文件就是这个类。下面就是这个类文件名和 FlashIDE 里的位置。

/source/projects/blastermines/flashIDE/com/efg/games/blastermines/Main.as

And this is for the Flex SDK (using Flash Develop):

/source/projects/blastermines/flexSDK/src/com/efg/games/blastermines/Main.as

这里就是 Main.as 类的整段代码。在我们看完这个类的所有代码之后，再看更改的地方。

```
package com.efg.games.blastermines
{
import com.efg.framework.FrameCounter;
import com.efg.framework.FrameRateProfiler;
import com.efg.framework.GameFrameWorkAdvancedTimer;
import flash.text.TextFormat;
import flash.text.TextFormatAlign;
import flash.geom.Point;
import flash.events.Event;
import com.efg.framework.FrameWorkStates;
import com.efg.framework.GameFrameWork;
import com.efg.framework.BasicScreen;
import com.efg.framework.ScoreBoard;
import com.efg.framework.SideBySideScoreElement;
import com.efg.framework.SoundManager;
```



```
public class Main extends GameFrameWork {
//custom sccore board elements
public static const SCORE_BOARD_SCORE:String = "score";
public static const SCORE_BOARD_LEVEL:String = "level";
public static const SCORE_BOARD_SHIELD:String = "shield";
public static const SCORE_BOARD_PARTICLE_POOL:String = "particlepool";
public static const SCORE_BOARD_PARTICLE_ACTIVE:String = "particleactive";
public static const SCORE_BOARD_PROJECTILE_POOL:String = "projectilepool";
public static const SCORE_BOARD_PROJECTILE_ACTIVE:String = "projectileactive";
//custom sounds
public static const SOUND_MINE_EXPLODE:String="SoundMineExplode";
public static const SOUND_MUSIC_IN_GAME:String="SoundMusicInGame";
public static const SOUND_MUSIC_TITLE:String="SoundMusicTitle";
public static const SOUND_PLAYER_ENTER:String = "SoundPlayerEnter";
public static const SOUND_PLAYER_EXPLODE:String="SoundPlayerExplode";
public static const SOUND_PLAYER_HIT:String="SoundPlayerHit";
public static const SOUND_PLAYER_SHOOT:String = "SoundPlayerShoot";
// Our construction only calls
public function Main() {
//added in chapter 11
if (stage) addToStage();
else addEventListener(Event.ADDED_TO_STAGE, addToStage,false,0,true);
}
//function added in chapter 11
override public function addToStage(e:Event = null):void {
if (e != null) {
removeEventListener(Event.ADDED_TO_STAGE, addToStage);
}
super.addToStage();
trace("in blastermines added to stage");
init();
}
// init() is used to set up all of the things that we should only need to do one time
override public function init():void {
trace("init");
game= new BlasterMines();
setApplicationBackGround(600, 400, false, 0x000000);
//add score board to the screen as the second layer
scoreBoard = new ScoreBoard();
addChild(scoreBoard);
}
```



```
scoreBoardTextFormat = new TextFormat("_sans", "11", "0xffffffff", "true");
scoreBoard.createTextElement(SCORE_BOARD_SCORE, new SideBySideScoreElement(450,
100, 20, "Score", scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_LEVEL, new SideBySideScoreElement(450,
120, 20, "Level", scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_SHIELD, new SideBySideScoreElement(450,
140, 20, "Shield", scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PARTICLE_POOL, new _
SideBySideScoreElement(450, 160, 20, "PartPool", scoreBoardTextFormat, _
50, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PARTICLE_ACTIVE, new _
SideBySideScoreElement(450, 170, 20, "PartActive", scoreBoardTextFormat, _
50, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PROJECTILE_POOL, new `CCC
SideBySideScoreElement(450, 180, 20, "ProjPool", scoreBoardTextFormat, _
50, "0", scoreBoardTextFormat));
scoreBoard.createTextElement(SCORE_BOARD_PROJECTILE_ACTIVE, new _
SideBySideScoreElement(450, 190, 20, "ProjActive", scoreBoardTextFormat, _
50, "0", scoreBoardTextFormat));
//screen text initializations
screenTextFormat = new TextFormat("_sans", "16", "0xffffffff", "false");
screenTextFormat.align = flash.text.TextFormatAlign.CENTER;
screenButtonFormat = new TextFormat("_sans", "12", "0x000000", "false");
titleScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE,400,400,_
false,0x000000 );
titleScreen.createOkButton("Play", new Point(150, 250), 100, 20, _
screenButtonFormat, 0x000000, 0xff0000,2);
titleScreen.createDisplayText("Blaster Mines", 200, new Point(100, 150), _
screenTextFormat);
instructionsScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS,_
400,400,false,0x000000);
instructionsScreen.createOkButton("Start", new Point(150, 250), 100, 20,_
screenButtonFormat, 0x000000, 0xff0000,2);
instructionsScreen.createDisplayText("Shoot everything\nDon't get hit.",_
200,new Point(100,150),screenTextFormat);
gameOverScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER,_
400,400,false,0x0000dd);
gameOverScreen.createOkButton("Restart", new Point(150, 250), 100, 20,_
screenButtonFormat, 0x000000, 0xff0000,2);
gameOverScreen.createDisplayText("Game Over",100,new Point(150,150),_
```



```
screenTextFormat);
levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN, _
400, 400, true, 0xbbff00ff);
levelInText = "Level ";
levelInScreen.createDisplayText(levelInText,100,new Point(150,150),_
screenTextFormat);
pausedScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_PAUSE,400,400,_
false,0xff000000 );
pausedScreen.createOkButton("UNPAUSE", new Point(150, 250), 100, 20, _
screenButtonFormat, 0x000000, 0xff0000,2);
pausedScreen.createDisplayText("Paused", 200, new Point(100, 150), _
screenTextFormat);
//set initial game state
switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
//sounds
//*** Flex SDK
soundManager.addSound(SOUND_MINE_EXPLODE,new Library.SoundMineExplode);
soundManager.addSound(SOUND_MUSIC_IN_GAME, new Library.SoundMusicInGame);
soundManager.addSound(SOUND_MUSIC_TITLE,new Library.SoundMusicTitle);
soundManager.addSound(SOUND_PLAYER_ENTER,new Library.SoundPlayerEnter);
soundManager.addSound(SOUND_PLAYER_EXPLODE,new Library.SoundPlayerExplode);
soundManager.addSound(SOUND_PLAYER_HIT,new Library.SoundPlayerHit);
soundManager.addSound(SOUND_PLAYER_SHOOT,new Library.SoundPlayerShoot);
//flash IDE
//soundManager.addSound(SOUND_MINE_EXPLODE,new SoundMineExplode);
//soundManager.addSound(SOUND_MUSIC_IN_GAME, new SoundMusicInGame);
//soundManager.addSound(SOUND_MUSIC_TITLE,new SoundMusicTitle);
//soundManager.addSound(SOUND_PLAYER_ENTER,new SoundPlayerEnter);
//soundManager.addSound(SOUND_PLAYER_EXPLODE,new SoundPlayerExplode);
//soundManager.addSound(SOUND_PLAYER_HIT,new SoundPlayerHit);
//soundManager.addSound(SOUND_PLAYER_SHOOT,new SoundSkullHit);
//framerate profiler
frameRate = 40;
frameRateProfiler = new FrameRateProfiler();
frameRateProfiler.profilerRenderObjects = 4000;
frameRateProfiler.profilerRenderLoops = 7;
frameRateProfiler.profilerDisplayOnScreen= true;
frameRateProfiler.profilerXLocation = 0;
frameRateProfiler.profilerYLocation = 0;
addChild(frameRateProfiler);
```



```
frameRateProfiler.startProfile(frameRate);
frameRateProfiler.addEventListener(FrameRateProfiler.EVENT_COMPLETE, _
frameRateProfileComplete, false, 0, true);
}
override public function frameRateProfileComplete(e:Event):void {
trace("profiledFrameRate=" + frameRateProfiler.profilerFrameRateAverage);
game.setRendering(frameRateProfiler.profilerFrameRateAverage, frameRate);
game.timeBasedUpdateModifier = frameRate;
removeEventListener(FrameRateProfiler.EVENT_COMPLETE, frameRateProfileComplete) ;
removeChild(frameRateProfiler);
//frame counter
frameCounter.x = 400;
frameCounter.y = 200;
frameCounter.profiledRate = frameRateProfiler.profilerFrameRateAverage;
frameCounter.showProfiledRate = true;
addChild(frameCounter);
startTimer(true);
}
override public function systemGamePlay():void {
game.runGameTimeBased(paused,timeDifference);
}
override public function systemTitle():void {
soundManager.playSound(SOUND_MUSIC_TITLE, true,999, 20, 1);
super.systemTitle();
}
override public function systemNewGame():void {
trace("new game");
soundManager.stopSound(SOUND_MUSIC_TITLE,true);
super.systemNewGame();
}
override public function systemLevelIn():void {
levelInScreen.alpha = 1;
super.systemLevelIn();
}
override public function systemWait():void {
//trace("system Level In");
if (lastSystemState == FrameworkStates.STATE_SYSTEM_LEVEL_IN) {
levelInScreen.alpha -= .01;
if (levelInScreen.alpha < 0 ) {
dispatchEvent(new Event(EVENT_WAIT_COMPLETE));
```



```
levelInScreen.alpha = 0;
}
}
}
}
}
```

我们已经对 Blaster Mines 的 Main.as 类做了很多改变。屏幕和声音非常简单，和前面我们的做的游戏里的改动很相似。

我们会花一点时间再那个上面，然后集中在添加暂停，静音，基于时间的步进计时器，FrameRateProfiler 和 FrameCounter 功能的必要修改上。我们需要下面的基本屏幕。

\_ Title screen: 包含了一个 Play 按钮和 Blaster Mines 文字。

\_ Instructions screen: 包含了一个开始按钮和 Shoot everything , Don' t get it 文字。

\_ Game-over screen: 包含了一个 Game Over 和一个 Restart 按钮

我们也需要下面这些 BasicScreen 类的实例。

\_ Level-in screen: 仅仅包含 Level 文字和 level 变量

\_ Paused screen: 包含 Paused 这样的文字和一个按钮，点击后可以取消系统的暂停。下面是我们需要对 ScoreBoard 类进行得修改。

\_ Score indicator

\_ Level indicator

\_ Shield indicator

\_ Particle pool indicators

\_ Projectile pool indicators

我们在变量声明的地方也需要几个新的常量来为新的 socredBoard 元素和声音服务。这些将添加在变量声明的地方。

```
//custom sscore board elements
public static const SCORE_BOARD_SCORE:String = "score";
public static const SCORE_BOARD_LEVEL:String = "level";
public static const SCORE_BOARD_SHIELD:String = "shield";
public static const SCORE_BOARD_PARTICLE_POOL:String = "particlepool";
public static const SCORE_BOARD_PARTICLE_ACTIVE:String = "particleactive";
public static const SCORE_BOARD_PROJECTILE_POOL:String = "projectilepool";
public static const SCORE_BOARD_PROJECTILE_ACTIVE:String = "projectileactive";
//custom sounds
```



```
public static const SOUND_MINE_EXPLODE:String="SoundMineExplode";
public static const SOUND_MUSIC_IN_GAME:String="SoundMusicInGame";
public static const SOUND_MUSIC_TITLE:String="SoundMusicTitle";
public static const SOUND_PLAYER_ENTER:String = "SoundPlayerEnter";
public static const SOUND_PLAYER_EXPLODE:String="SoundPlayerExplode";
public static const SOUND_PLAYER_HIT:String="SoundPlayerHit";
public static const SOUND_PLAYER_SHOOT:String = "SoundPlayerShoot";
```

## 实现暂停和静音功能

我们前面已经讨论过暂停和静音功能，但是在 Main.as 类里我们现在必须修改 init 函数并且添加进一个复写过的 addToStage 功能。在我们知道 Flash 舞台可用后，这些就是我们能为暂停和静音功能添加键盘的监听的必要条件。注意一点，如果你还没有这么做，你需要添加这一行

```
public static const STATE_SYSTEM_PAUSE:int = 99;
```

到 com.efg.framework.FrameWorkStates.as 文件。

## 添加新的构造函数

我们用下面这个替换到 Main.as 里当前的构造函数

```
public function Main() {
//added in chapter 11
if (stage) addToStage();
else addEventListener(Event.ADDED_TO_STAGE, addToStage,false,0,true);
}
```

新的构造函数在舞台可用之前设置一个监听，当舞台可用（或者在构造函数第一运行的时候已经可用了），调用 addToStage 函数

## 添加 addToStage 函数

这个函数将会复写 GameFrameWork.as 版本的 addToStage 函数然后调用 super

```
override public function addToStage(e:Event = null):void {
if (e != null) {
removeEventListener(Event.ADDED_TO_STAGE, addToStage);
}
super.addToStage();
trace("in blastermines added to stage");
init();
}
```

这个函数主要的目的是在舞台对 SWF 文件可用之后添加一个舞台的引用。我们将会通过 super.addToStage 调用 GameFrameWork 类的 addToStage 函数和我们游戏的 init 函数。



注意，我们已经将调用的地方从 init 函数移动到了 addToStage 函数。这是为了保证直到 GameFramework 的 addToStage 函数已经被调用之前，我们不会试着初始化游戏。

接下来，我们需要添加 pausedScreen。这个 BasicScreen 框架类的实例，已经在 GameFramework.as 文件中创建了，但是在 Main 中实例化并自定义颜色的大小的。你也可以用 addChild 添加任何你认为你需要添加到屏幕上的自定义元素；它不一定非要只是一个拥有背景颜色，一个按钮和一小撮文字的简单屏幕。这些建议同样适用于 BasicScreen 的所有实例。

```
pausedScreen = new
BasicScreen(FrameWorkStates.STATE_SYSTEM_PAUSE,400,400,false,0xff000000 );
pausedScreen.createOkButton("UNPAUSE", new Point(150, 250), 100, 20,
screenButtonFormat, _
0x000000, 0xff0000,2);
pausedScreen.createDisplayText("Paused", 200, new Point(100, 150), screenTextFormat);
```

暂停功能需要将 GameFramework 类的 paused 变量传入 Game 类的实例。让我们看一看添加基于时间的步进计时器功能需要做什么和怎么做。

## 在 Blaster Mines 实现基于时间的步进计时器

我们需要去复写 systemGamePlay 功能，将 paused 和 timeDifference 变量传入 Game 类。

```
override public function systemGamePlay():void {
game.runGameTimeBased(paused,timeDifference);
}
```

我们前面提到了，Game 基类会添加一个 runGameTimeBased 函数。这个函数将会在这两个变量上工作。同样的，startTimer 函数的调用已经被移到新的 frameRateProfileComplete 函数了。下面让我们开始讨论 FrameRateProfiler 的实现。

## 在 Blaster Mines 中自定义 FrameRateProfiler

我们要为这个游戏定制一个 FrameRateProfiler 就需要再 init 函数里添加一些新的代码。

```
frameRate = 40;
frameRateProfiler = new FrameRateProfiler();
frameRateProfiler.profilerRenderObjects = 4000;
frameRateProfiler.profilerRenderLoops = 7;
frameRateProfiler.profilerDisplayOnScreen= true;
frameRateProfiler.profilerXLocation = 0;
frameRateProfiler.profilerYLocation = 0;
```

首先，我们给我们游戏置一个期望的 frameRate。这个值不会大于你在 swf 的 publish 参数里设定的帧率。profileRenderObjects 值是我们碰到的在 FrameRateProfiler 里最重要的一个设定。我们需要对这个数字进行一个小小的实验，来看他如何影响渲染分析。原因是每台机器和插件版本不同得出的



FrameRateProfiler 不尽相同。出于这个原因，对于你自己的或者你可用的机器，你需要基于对象的数量进行分析。你需要校准你自己开发环境的分析器。

我们如何校准 profileRenderObjects 值呢？例如，如果我们想要游戏运行在 40 帧率，我们就需要设置分析器的对象数量到一个任意的数量（我们总是说 4000）然后运行分析器和游戏。如果分析器的帧率结果出来是 35，但是当我们运行游戏的时候我们的电脑能稳定在 40 帧左右，我们就需要减少我们分析器里的对象。因为这表示分析器里的对象数量过高，这样使我们的分析器负担过重以至于不能体现出我们的游戏性。你也许会玩这个游戏很多次，在玩的过程中观察帧率。目的是让分析的帧率和你机器上游戏中运行相符。这种校正是必须的，以保证分析器里的对象数目能够很好的体现，广大玩家的机器运行我们游戏的情况。

基于时间的步进计时器将会保证所有种类的机器，游戏对象运行在同样的速度上。我们也使用分析过的帧率作为参数传递给 Game 类的 setRendering 函数。这将允许我们根据用户机器的分析结果来，增加或者减少游戏效果。FrameRateProfiler 的最后三条，添加到舞台显示列表，启动它，添加一个完成时的监听器。

```
addChild(frameRateProfiler);
frameRateProfiler.startProfile(frameRate);
frameRateProfiler.addEventListener(FrameRateProfiler.EVENT_COMPLETE, _
frameRateProfileComplete, false, 0, true);
```

当 profile 完成时候 frameRateProfileComplete 函数就别调用了。让我们现在来看一下这个函数。

## 创建 frameRateProfileComplete 函数

FrameRateProfiler 完成它的工作后，就会调用 frameRateProfileComplete 函数。让我们一行一行的看看它是怎么工作和完成了什么工作。当 profile 完成的时候，我们做的第一件事情就是调用我们在 Game 类里添加的新函数 setRendering。我们将 profilerAverageFrameRate 和游戏的设计帧率一起传进去。如果需要，我们能使用这两个值设置自定义的游戏效果质量。我们将会在 Blaster Mines 游戏里这样做。

```
game.setRendering(frameRateProfiler.profilerFrameRateAverage, frameRate);
game.timeBasedUpdateModifier = frameRate;
```

这两行，我们只是简单的移除了 FrameRateProfiler 并且删除了事件监听。

```
removeEventListener(FrameRateProfiler.EVENT_COMPLETE, frameRateProfileComplete);
removeChild(frameRateProfiler);
```

这里的五行将可选的 FrameCounter 实例放到舞台上。

```
//frame counter
frameCounter.x = 400;
frameCounter.y = 200;
frameCounter.profiledRate = frameRateProfiler.profilerFrameRateAverage;
frameCounter.showProfiledRate = true;
addChild(frameCounter);
```



最后，我们启动游戏计时器。传入一个 true，这样我们就是用了新的基于时间的步进计时器，而不是在第二章中创建的原始计时器。

```
startTimer(true);
```

## 创建 Library.as 类

Library.as 类文件仅对那些使用 Flex SDK 框架的是必要的，Flash IDE 工程不需要使用它。库的修改主要集中在讲声音作为静态常量资源添加进来，而不是放在 SWF 从 IDE 中导出。这样做，我们的游戏将完全不依靠 IDE。解耦了游戏和 IDE 的缺点是 Flex 框架导入.wav 文件很无力。借助.mp3 文件我们能缓和这种局限，但是随之而来的问题是尝试去避免.mp3 文件头部的安静的部分。播放声音的时候我们将会使用偏移量设定来跳过无声的部分。我们将在第十章里更仔细的讨论这个限制。当同样的，添加 Drive She Said 声音作为 MP3 文件到这个库。

注意如果你使用 FlashIDE 你需要导入所有的 mp3 声音并且创建和 Library 类里类名一样的连接名。

作为 Blaster Mines 游戏包的一下一个新文件，让我们来创建这个类吧。这里类文件名和位置。并且这是 FlexSDK ( 使用 Flash Develop ) 的：

```
/source/projects/blastermines/flexSDK/src/com/efg/games/blastermines/Library.as
package com.efg.games.blastermines
{
    public class Library {
        [Embed(source='.././.././../assets/mineExplode.mp3')]
        public static const SoundMineExplode:Class;
        [Embed(source='.././.././../assets/musicIngame.mp3')]
        public static const SoundMusicInGame:Class;
        [Embed(source='.././.././../assets/musicTitle.mp3')]
        public static const SoundMusicTitle:Class;
        [Embed(source='.././.././../assets/playerEnter.mp3')]
        public static const SoundPlayerEnter:Class;
        [Embed(source='.././.././../assets/playerExplode.mp3')]
        public static const SoundPlayerExplode:Class;
        [Embed(source='.././.././../assets/playerHit.mp3')]
        public static const SoundPlayerHit:Class;
        [Embed(source='.././.././../assets/playerShoot.mp3')]
        public static const SoundPlayerShoot:Class;
    }
}
```

注意，到目前为止和所有的库类构造函数一样，我们需要保证声音的类名和 Main.as 里的一致，并且使用正确的资源文件夹路径。

## 修改 SoundManager 类



要让静音功能工作正常需要修改游戏框架里的 SoundManager 类。记住，这个在 com.efg.framework 包里。我们需要添加两个私有变量：Private var soundMute:Boolean 和 private var

```
muteSoundTransform:SoundTransform = new SoundTransform();
```

当 muteSound 函数被调用的时候，布尔类型私有变量 soundMute 会由 true 变成 false，反之亦然。

private var muteSoundTransform:SoundTransform = new SoundTransform();变量是一个 SoundTransform 类的实例，经常被用来设置音量，从 0( soundMute=true)到 100( soundMute=false )。

现在，让我们看一看 muteSound 公共函数。

```
public function muteSound():void {  
    //trace("sound manager got mute event");  
    if (soundMute) {  
        soundMute=false;  
        muteSoundTransform.volume=1;  
        SoundMixer.soundTransform=muteSoundTransform;  
    }else{  
        muteSoundTransform.volume=0;  
        SoundMixer.soundTransform=muteSoundTransform;  
        soundMute=true;  
    }  
}  
  
import flash.media.SoundMixer;
```

注意，通过设定 soundTransform 属性为我们自己的 muteSoundTransform 变量，我们设置了全局的 SoundMixer 的音量。我们必须 import 进 SoundMixer 类来这样做。添加这一行到这个类的 import 部分。

在我们开始看一看我们打算添加到游戏框架和 Blaster Mines 包的新的游戏性相关类的细节。让我们看一看更多在我们游戏里实现的优化理论。

## 对象池优化

对象池是一种帮助节省系统内存和处理器执行时间的优化技术。我们将创建一个我们对象类型的数组，我们叫‘池（pool）’，来作为缓存对象的方法。当我们需要使用一个对象的时候，我们将会从池中去取。如果池中没有可用对象，我们就不能显示一个对象。这对象粒子一样的特效非常适合，不会影响到游戏性。

## 节省处理执行时间

创建一个新的对象是一项运算量非常大的活动。在需要对象时，减少了对对象每次实例化需要，我们就减少了整个游戏的执行时间。

## 节省内存



对象池为什么能节省内存不是那么明显，因为我们要创建一个全局对象池，而它使用系统内存并且直到我们销毁它们之前它不会被垃圾回收。在游戏开始的时候使用的内存更多了，因为要填满对象池并且使它准备好。我们用这个技术节省内存效果是双重的和微秒的。

首先，如果我们设置池的大小来限制总量，我们可以组织系统创建太多的对象，用完所有可用的内存。这也能通过其他方式实现，如设定一个最大值，每次创建对象的时候检查一下，但是这个方法也对池适用。

第二，对象池帮助平缓系统的垃圾回收处理。如果我们经常创建和回收上千个例子（既实例），在真正被垃圾回收之前将会消耗一定的时间。如果我们销毁 100 个对象然后下个粒子爆炸又创建 100 个，没有对象池我们将会使用一倍的内存，因为垃圾回收处理没必要为下一组实例化而即时的释放销毁对象的内存。

如果我们用池来缓存对象，垃圾回收器需要做的就少得多，并且不会因为垃圾回收处理开始并且移除所有不要的未缓存的对象，而使渲染经历一个大停顿。

## 在游戏里实现对象缓存

我们将在玩家发射的导弹和爆炸粒子中使用对象池。游戏设置了每个对象池的基本数量。如果 FrameRateProfiler 的 frameRateProfilerAverage 能运行在设计帧率的 85% 或者更多 粒子池就要扩大一倍。这只是一个简单的例子，关于我们如何能结合渲染分析和其他的优化来影响指定的玩家系统上游戏的整体表现

注意你也可以使用分析过的帧率，作为一个设置 stage.quality 属性设置为 low, medium, high 或者 best 的一个指标。这个影响到舞台上绘制矢量和位图的质量。我们将会在 Blaster Mines 游戏里使用。

## 创建 Blaster Mines 对象缓冲的技术规格

我们将会在我们游戏里，为对象池创建独立的管理类。这些管理类将会封装对象缓冲需要的变量和缓冲对象的创建函数

玩家必须射击的 Mine 敌机也有一个管理类，但是我们没有必要为这些对象实现一个对象池。

作为一个例子，我们看一看 ParticleManager 类里缓冲对象的代码。这个类将在“设计粒子管理类”这一章节里完整的呈现出来。

## 添加私有变量

下面的私有变量会在 com.efg.games.blastermines.ParticleManager 类的变量定义部分创建：

\_ private var particleBitmapData:BitmapData: 保存了画进 BitmapData 容器的粒子外观

private var particleAnimationFrames:Array:保存了粒子的消失的动画帧。

\_ private var particles:Array:活动的粒子数组

\_ private var particlePool:Array:非活动的粒子数组

\_ private var particleCount:int:活动粒子数组的循环中使用

\_ private var particlePoolCount:int:非活动粒子数组的循环中使用



```
_ private var tempParticle:BasicBiltArrayParticle:循环中使用的临时变量  
_ private var particlePoolMax:int = 500:对象池中对象的总数  
_ private var particlesPerExplode:int = particlePoolMax / 20: 一个爆炸需要的粒子数
```

## 实例化一个池中的粒子

每关开始的时候调用 createParticlePool 函数，保证创建的粒子颜色符合本关的敌人的颜色。这个函数在 com.efg.games.blastermines.ParticleManager 类：

```
public function createParticlePool(maxParticles:int):void {  
    particlePool = [];  
    particles = [];  
    for (var particleCtr:int=0;particleCtr<maxParticles;particleCtr++) {  
        var tempParticle:BasicBiltArrayParticle = new BasicBiltArrayParticle(0,799,0,799);  
        particlePool.push(tempParticle);  
    }  
}
```

这个函数循环将粒子放入 particlePool 数组，循环的次数为传入的参数。

## 激活粒子

在游戏里，爆炸产生的时候，Mine 就被摧毁了。CreateExplode 函数其这个作用。这个函数是 com.efg.games.blastermines.BlasterMines 类的一部分。

```
private function createExplode(xval:Number,yval:Number,parts:int):void {  
    for (var explodeCtr:int=0;explodeCtr<parts;explodeCtr++) {  
        particleManager.particlePoolCount = particleManager.particlePool.length-1;  
        if (particleManager.particlePoolCount > 0) {  
            tempParticle=particleManager.particlePool.pop();  
            tempParticle.lifeDelayCount=0;  
            tempParticle.x=xval;  
            tempParticle.y = yval;  
            tempParticle.nextX=xval;  
            tempParticle.nextY=yval;  
            tempParticle.speed = (Math.random() * 3) + 1;  
            tempParticle.frame = 0;  
            tempParticle.animationList = particleManager.particleAnimationFrames;  
            tempParticle.bitmapData = tempParticle.animationList[tempParticle.frame];  
            var randInt:int = int(Math.random() * 359);  
            tempParticle.dx = rotationVectorList[randInt].x;  
            tempParticle.dy = rotationVectorList[randInt].y;  
            particleManager.particles.push(tempParticle);  
        }  
    }  
}
```



```
}  
}  
}
```

函数循环创建以传入的 xval 和 yval 坐标点为中心开始的爆炸 循环的次数为传入的参数 parts 的数值。

粒子将会随机选择一个移动的方向，然后从该点发射出去。

首先，createExplode 函数检查来确定池子里有可用的粒子实例。如果池子里没有粒子实例剩余，就不会创建粒子。如果可能，就从池中取出一个粒子，修改它的属性并且将他添加到粒子数组里。具体做法是，从 particlePool 数组里弹出下一个粒子并且将引用传给 tempParticle，这样就将 particlePool 数组里最后一个粒子的引用赋值给了 tempParticle。

## 粒子去活

当粒子生命结束时，就会从粒子数组移回到 particlePool 数组。这些发生在 BlasterMines.as 类的 update 函数。

```
particleManager.particleCount = particleManager.particles.length-1  
for (var ctr:int = particleManager.particleCount; ctr >= 0; ctr--) {  
    tempParticle = particleManager.particles[ctr];  
    if (tempParticle.update(step)) { //return true if particle is to be removed  
        tempParticle.frame = 0;  
        particleManager.particlePool.push(tempParticle);  
        particleManager.particles.splice(ctr,1);  
    }  
}
```

有时候，我们把单个处理优化的概念叫做“单对象池”。贯穿整部书，我们已经创建了数目众多的类级别变量，来作为处理的重用。“类级别”，意思是一个变量，有着全局的范围，每个类里的函数都能使用它。对于类似 Point 和 Rectangle 对象的实例我们倾向于这样做，因为他们会被多次创建和使用（对于每个 blitted 对象实例，30 到 40 次每秒）为创建一个使每个都能用，我们本质上会创建单个对象池。这个技术是有争议的，因为（和其他的对象池相比）节省创建这些对象的时间，当游戏初始化的时候，也使用了更多的内存，或者创建了更大的程序运行时内存。尽管同样的理论从对象池章节一直持续到这里。整体上来看，我们通过创建这些“单个对象池”减少了内存的消耗，因为对象数量的总和在任何一个时候都是恒定的，而不是不定时的垃圾回收处理时上下剧烈的摆动。如果我们在每帧创建局部对象则会出现这种情况。

贯穿整部书，在每个 ScoreBoard 类元素更新的时候，我们只创建了非常少的自定义事件调用。在 Blaster Mines，我们将会为每个这样的更新事件创建一个单独的重用事件。接下来让我们看看具体是怎么回事。

## 重用全局事件对象

BlasterMines.as 类在每帧里发送出了大量的事件来更新 ScoreBoard。我们通过重用为更新每个 ScoreBoard 元素而触发的事件，来降低程序运行时的内存，同样的也会降低处理和执行时间。



我们将以 score 元素为例，但是这会应用到我们 Game 类里所有的 ScoreBoard 显示元素上。

1.首先，我们需要为每次分数更新创建一个重用事件对象：

```
private var customScoreBoardEventScore:CustomEventScoreBoardUpdate = new _  
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate._  
UPDATE_TEXT,Main.SCORE_BOARD_SCORE, "");
```

2. 当我们在 score 上进行更新 Main.SCORE\_BOARD\_SCORE 文本元素的调用时，我们需要将事件的属性值设置为 score 值。

```
customScoreBoardEventScore.value = score.toString()
```

3.最后，我们需要发送事件：

```
dispatchEvent(customScoreBoardEventScore);
```

就像你看到的，每次我们更新 score 和创建新事件时，我们只是简单的重用同样的类级别的可变事件

## 优化查找表

超找表是进行复杂数学计算时一个非常简单的优化。这个有点争议，AS3 里 math 是不是比数组查找快一些，但是因为使用了 FrameCounter，在转动矢量值转动一定弧度的时候，使用一个查找表就能够看出每 1FPS

里的微小不同之处。如果应用到一个非常大型的应用里，你就能看到更大的改进。根据经验，我们使用 Vector 类作为我们的查找表会比使用数组作为查找表快。如果你的不是一个兼容 Flash Player10 的发布系统，你可以用一个数组实例替换 Vector。我们已经打算使用 BlitArrayAsset 类来创建 360 张玩家飞船图片的旋转数组。因为这个我们能够使用 BasicArrayBlitObject 的同样的帧属性来取出矢量值需要在我们的玩家的朝向上使用。

## 创建移动向量查找表

接下来这些步骤是用来创建移动查找表的

1.我们添加一个全局的数组或者矢量变量到 BlasterMines.as 类，保存 Point 对象实例。Point 对象里面则保存的是游戏对象旋转的方向我们需要移动的到 dx 和 dy 值。Point 的 x 存放 dx 的值，Point 的 y 存放 dy 的值。这样就比普通对象实例稍微少用一点内存，因为他是不可扩展和非动态的，使 Flash Player 避免了定位内存，因为为了以防普通对象实例运行时添加属性。所有的代码将在 com.efg.games.blastermines.BlasterMines.as 文件里。首先从变量定义部分看起。

```
//math look up tables  
//private var rotationVectorList:Array = [];  
private var rotationVectorList:Vector.<Point> =new Vector.<Point>(360,false);
```

2.BlasterMines.as 类初始化的部分调用一个函数来创建查找表

```
createLookupTables();
```



3.createLookUpTables 函数循环计算从 0 到 359 度，移动一个物体需要的 dx 和 dy 矢量值。

```
private function createLookUpTables():void {  
    for (var ctr:int = 0; ctr < 359; ctr++) {  
        var point:Point = new Point();  
        point.x = Math.cos((Math.PI * ctr) / 180);  
        point.y = Math.sin((Math.PI * ctr) / 180);  
        rotationVectorList[ctr] = point;  
    }  
}
```

当一个游戏对象需要按照一定的角度移动到一个特殊的方向，就要使用 rotationVectorList 代替临时计算 dx 和 dy。在下一章节我们会看到一个这样的例子。

## 访问 vectorRotationList 查找表

在这个游戏里所有的显示对象都会按照一定的角度移动。让我们看一看 Projectile 对象实例如何被创建和访问这个表。

玩家发射的导弹，每帧自动发射，每次发射后有 3 帧的延时。发射的物体将会像粒子一样被缓冲。在 BlasterMines.as 的 update 函数里创建发射的物体。下面就是实现这段功能的代码;这段代码在 com.efg.games.blastermines.BlasterMines.as 文件里。

```
/** auto fire  
projectileManager.projectilePoolCount = projectileManager.projectilePool.length - 1;  
if (projectileManager.lastProjectileShot > 3 && projectileManager.projectilePoolCount _  
> 0 && playerStarted && mineManager.mineCount > 0) {  
    dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _  
Main.SOUND_PLAYER_SHOOT, false, 0, 8, 1));  
    tempProjectile = projectileManager.projectilePool.pop();  
    var projectileRadians:Number = (player.frame / 360) * 6.28;  
    //+ 16 to get it to the center of a 32x32 sprite  
    tempProjectile.x=(player.point.x+16)+Math.cos(projectileRadians);  
    tempProjectile.y =(player.point.y+16) + Math.sin(projectileRadians);  
    tempProjectile.x = player.x+16;  
    tempProjectile.y = player.y + 16;  
    tempProjectile.nextX = tempProjectile.x;  
    tempProjectile.nextY = tempProjectile.y;  
    tempProjectile.dx = rotationVectorList[player.frame].x;  
    tempProjectile.dy = rotationVectorList[player.frame].y;  
    tempProjectile.speed = 5;  
    tempProjectile.frame = 0;  
    tempProjectile.bitmapData = tempProjectile.animationList[0];
```



```
projectileManager.projectiles.push(tempProjectile);
projectileManager.lastProjectileShot=0;
}else {
projectileManager.lastProjectileShot+=step;
}
```

前面的函数是实际上的全部开火发射代码。我们只需要注意这两行：

```
tempProjectile.dx = rotationVectorList[player.frame].x;
tempProjectile.dy = rotationVectorList[player.frame].y;
```

玩家对象当前帧索引是 player.animationList 数组里旋转过的 BitmapData，表示玩家当前旋转的角度。炮弹使用相同的索引值将他们的 dx 和 dy 值从 rotationVectorList 数组中取出，x 值就是 dx，y 值就是 dy。

## 使用 ScrollRect 优化基于屏幕的 blit 滚屏

AS3 编程里有太多的方法滚屏了。在前一个章节里，我们看到了基于 tile 的 blit 渲染的 360 度滚屏。在这一章里，我们将会看到一个优化过的方法来滚屏，不需要按照方块来（虽然可以）。我们将会以矢量图形的方式绘制所有游戏资源，添加一些简单的发光滤镜，并且缓存这些 BitmapData。我们会绘制这些游戏对象到一个整个游戏世界大小的 Bitmapdata 容器里，但是仅仅显示当前窗口可见的部分给用户。滚屏使用的一个和第 10 章的 Camera2D 类不同的方法。按照我们的经验，在 AS3 里滚动一个位图屏幕最快的方法是使用世界容器 Bitmap 实例的 scrollRect 属性。我们简单的修改一下这个 Bitmap 的 scrollRect 矩形的左上角的 x 和 y 坐标。在第十章里，我们使用了一个 copyPixels 偏移的方式在世界容器里滚动可视的窗口。在使用 scrollRect 方法替代 copyPixels 方法时，我们在 Blaster Mines 里测试，注意到了每秒 2 到 3 帧的速度增加。这个方法也大大的简化并且降低了滚屏必须的操作的次数。在 Blaster Mines 里，整个世界是 800\*800，可视部分是 400\*400。我们将不会使用 Drive She Said 里的 tiles，所以我们不需要缓存额外的 tiles 来保证平滑滚动。我们所需要做的是：

- 1.根据鼠标的位置，更新玩家的 nextX 和 nextY 值

- 2.玩家已经根据他现在的位置被渲染到了 800\*800 的 canvasBitmapData 世界里。将他的 x 和 y 坐标更新成 nextX 和 nextY 一样的坐标值。

- 3.canvasBitmap 的左上角（这个显示对象保存了我们 canvasBitmapData 的 blit 容器）设置成 player.x-200,player.y-200。

大小始终设置成 400\*400,。这样我们的玩家就被放在了屏幕中央。

- 4.当玩家移动的时候，我们简单的改变这个 scrollRect 矩形的左上角的 x 和 y 坐标。这样就会看上去像在整个世界里滚屏。

```
if (playerStarted) {
canvasRect.x = player.x - 200;
canvasRect.y = player.y - 200;
if (canvasRect.x < 0) canvasRect.x = 0;
```



```
if (canvasRect.y < 0) canvasRect.y = 0;
if (canvasRect.x > 399) canvasRect.x = 399;
if (canvasRect.y > 399) canvasRect.y = 399;
canvasBitmap.scrollRect = canvasRect;
}
```

为了保证 canvasBitmap.scrollRect 不会移出游戏世界的 canvasBitmapData 的边界，我们确保左上角 x 和 y 方向都在 0 到 399 之间。

注意，我们更新一个叫 canvasRect 的矩形实例而不是直接更新 canvasBitmap.scrollRect。scrollRect 属性不能直接修改。你需要修改第二个矩形实例的属性然后将 canvasBitmap.scrollRect 赋值为 canvasRect（代码最后一行）。这个简单的滚屏技术十分有用，和我们说的一样，相比第 10 章使用的摄像机缓存方法每秒能节省 2~3 帧

## 雷达屏幕的 BitmapData 重用优化

Blaster Mines 游戏在游戏屏幕的右边有一个游戏世界整个的缩小版，创建起来非常简单，使用全屏 blitting 技术。

首先，我们讨论一下传统 Flash 方法滚屏和创建一个关联的雷达样式的屏幕。如果选择使用一个 Sprite 容器而不是一个 blit 容器，我们将用一个 Sprite 持有我们游戏屏幕的上千个其他附加在显示列表里的独立 Sprites(或者是 Bitmap 对象)。

我们可以使用 Sprite 持有者的实例的 scrollRect 属性滚动屏幕。尽管如此，我会失去世界 BitmapData 容器 canvasBitmapData 的可重用性。我们希望更新 radar 的每帧里，使用可滚动的 Sprite 容器和附加到显示列表的独立的 Sprite 对象，我们可以绘制 Sprite 容器的内容到一个叫 radarBD 的 BitmapData 上。或者我们用第二个 Sprite 对象创建一个缩小版本的 Sprite 容器来表现每个 Sprite 容器对象。所有的这些想法实现都没问题，但是和将 canvasBitmapData 作为我们 radarBitmap 的 BitmapData 引用的重用相比，他们会很慢。

明显的，使用第二个 Bitmap 作为一个雷达对象来显示整个游戏世界，只有在你希望雷达屏幕是真实游戏的缩小版时才起作用。如果雷达是游戏屏幕的抽象表现，你就会使用不同的方法。

当我们创建我们自己的 radarBitmap，我们把他的引用设置为 canvasBitmapData。

```
private var radarBitmap:Bitmap = new Bitmap(canvasBitmapData);
```

只要 canvasBitmapData 已经创建了，这段代码就能正常运作。在 init 函数里，我们简单的修改了 radarBitmap 并且放置在了屏幕上。然后我们就永远不需要再担心了。canvasBitmapData 更新了，他就会更新。

```
radarBitmap.x = 420;
radarBitmap.y = 230;
radarBitmap.scaleX = .2;
radarBitmap.scaleY = .2;
addChild(radarBitmap);
```



## 创建新游戏的类

接下来的类，尽管是为这个特定的游戏的需要创建的，也被设计成一个普通小片，这样就能被需要他们的游戏重用了。

一些将会被加入框架，并且一些会在 Blaster Mines 包里。

## 设计 BlitArrayAsset 类

BlitArrayAsset 类在 com.efg.framework 并且等同于在代码中创建的资源 TileSheet。这个辅助类只用来创建资源数组。对那些我们创建它们函数而言，这个类的实例总是临时并且只是局部的。

```
public function createRotationBlitArrayFromBD(sourceBitmapData:BitmapData, inc:int, _
offset:int = 0):Array {
```

这个函数需要传入一个 BitmapData(sourceBitmapData)实例，一个自增值和一个创建自转用角度偏移量。

BitmapData 表现为一张静态图片，这张图片会被转化成一个循环装有 BitmapData 值的数组。自增值表现为在每个自转的物体上滑过的度数。例如，一个 1 的增加值将会创建一个 360 度的数组（每 1 度的偏移量），当增加量的值为 10 将会创建一个 36 度的数组（每 10 度的偏移量）。

偏移值是用来保证我们对象的旋转值和对象的外表一致。你会注意到（在本章的后面一点），玩家的飞船是绘制成朝上的。我们希望这是 0 度值。Flash 实际上用朝右为 0 度值。传入 90 的偏移值后，我们就能减轻这种操作并且为我们的对象创建正确的角度值。这样，我们的值就会跟预计算的向量查找表中的相符，这些值就是创建来优化对象移动的。

如果图像需要绘制成向右而不是向上，这个偏移值就不需要了。我们会将游戏的图片都绘制成朝上的，因为我们发现通过这样语法的代码绘制更简单。这个偏移量同样允许使用指向任何方向的图片作为旋转数组的源数据。

这个功能使用一个 Matrix，首先转换这个 BitmapData（移动到一个位置，x 为宽度的-1/2,y 为高度的-1/2），按照新的自增量旋转初始值，再转换到原来的位置上。它为每个旋转，使用初始的 sourceBitmapData，来保证歪曲和位图的退化最低。Matrix 的代码是这样的：

```
var angleInRadians:Number = Math.PI * 2 * (rotation / 360);
var rotationMatrix:Matrix = new Matrix();
rotationMatrix.translate(-sourceBitmapData.width*.5,-sourceBitmapData.height*.5);
rotationMatrix.rotate(angleInRadians);
rotationMatrix.translate(sourceBitmapData.width*.5,sourceBitmapData.height*.5);
Our second public function is
public function createFadeOutBlitArrayFromBD(sourceBitmapData:BitmapData,_
steps:int):Array{
```

就像 createRotationBlitArrayFromBD 函数，这个函数以 sourceBitmapData 为参数，创建一个消失数组（透明通道）BitmapData 资源。步骤的值表现出消失动画的帧数。例如，如果传入的值为 10，这个函数每次迭代就会降低传入的 BitmapData（sourceBitmapData）的 alpha，降低值为 0.1。



这个函数使用一个 ColorMatrixFilter 实例来对 sourceBitmapData 制造一个 alpha 衰减。Matrix 操作的完整描述不在本书范围内，但是下面的 Matrix，应用到 sourceBitmapData 的 alpha 值上就会产生一个消失效果。

```
var alpha:Number=1 - (ctr*stepAmount)
var alphaMatrix:ColorMatrixFilter = new ColorMatrixFilter(
[1, 0, 0, 0, 0,
0, 1, 0, 0, 0,
0, 0, 1, 0, 0,
0, 0, 0, alpha, 0]);
```

这里是这个类的完整源码：

```
package com.efg.framework
{
import flash.display.*;
import flash.geom.*;
import flash.filters.ColorMatrixFilter;
public class BlitArrayAsset {
public var tileList:Array;
private var point0:Point = new Point(0, 0);
public function createRotationBlitArrayFromBD(sourceBitmapData:BitmapData, _
inc:int, offset:int = 0):Array {
tileList = [];
var rotation:int = offset;
while (rotation<(360+offset)){
var angleInRadians:Number = Math.PI * 2 * (rotation / 360);
var rotationMatrix:Matrix = new Matrix();
rotationMatrix.translate(-sourceBitmapData.width*.5,-sourceBitmapData.height*.5);
rotationMatrix.rotate(angleInRadians);
rotationMatrix.translate(sourceBitmapData.width*.5,sourceBitmapData.height*.5);
var matrixImage:BitmapData = new BitmapData(sourceBitmapData.width, _
sourceBitmapData.height, true, 0x00000000);
matrixImage.draw(sourceBitmapData, rotationMatrix);
tileList.push(matrixImage.clone());
rotation += inc;
matrixImage.dispose();
matrixImage = null;
rotationMatrix = null;
}
return(tileList);
}
```



```

}
public function createFadeOutBlitArrayFromBD(sourceBitmapData:BitmapData, _
steps:int):Array{
var stepAmount:Number = 1 / steps;
tileList = [];
for (var ctr:int = 0; ctr <= steps; ctr++) {
var alpha:Number=1 - (ctr*stepAmount)
var alphaMatrix:ColorMatrixFilter = new ColorMatrixFilter(
[1, 0, 0, 0, 0,
0, 1, 0, 0, 0,
0, 0, 1, 0, 0,
0, 0, 0, alpha, 0]);
var matrixImage:BitmapData = new BitmapData(sourceBitmapData.width, _
sourceBitmapData.height, true, 0x00000000);
matrixImage.applyFilter(sourceBitmapData, matrixImage.rect, point0, _
alphaMatrix);
tileList.push(matrixImage.clone());
matrixImage.dispose();
matrixImage = null;
alphaMatrix = null;
}
return(tileList);
}
}
}
}

```

createRotationBlitArray 和 createFadeOutBlitArray 方法你已经知道 将会用在保存我们的游戏对象的外观的 BitmapData 对象数组里。

下一步，我们看一下我们所有游戏对象的基类。这个基类就是 BasicBlitArrayObject。在下一节里我们会讨论到的管理类里，BliteArrayAsset 类将会用来创建这些 BasicBliteArrayObject 的外观。

## 设计 BasicBlitArrayObject 类

这章里，为替代游戏里的 tile，我们将会代码里绘制我们的图片并且使用一个数组的 BitmapData 实例来模拟动画。BasicBlitArrayObject 是基类，播放起动画就像是一个数组的 BitmapData 对象。它包含了 4 个公共函数和一些公共属性，这些属性在移动对象和播放一个数组的 BitmapData 对象动画时是必须的。这个类被添加在 framework 包里。

包的位置在 com.efg.framework

这些公共属性：

\_ public var x:Number: blitting canvasBitmapData 对象的 x 坐标（左上角）



---

- \_ public var y:Number: blitting canvasBitmapData 对象的 y 坐标 ( 左上角 )
- \_ public var nextX:Number: 在游戏的更新环节, 更新并且应用到渲染器里的 x 的值
- \_ public var nextY:Number:在游戏的更新环节, 更新并且应用到渲染器里的 y 的值
- \_ public var dx:Number:在更新环节, 应用到 nextX 的 x 的变更值
- \_ public var dy:Number:在更新环节, 应用到 nextY 的 y 的变更值
- \_ public var frame:int: animationList 数组的当前索引
- \_ public var bitmapData: animationList[frame]描绘的一个 BitmapData 的局部引用
- \_ public var animationList:Array: BitmapData 的局部数组的引用, 用来循环播放动画
- \_ public var point:Point: copyPixels 操作的全局点引用
- \_ public var speed:Number:每次更新加到 dx 和 dy 的值
- \_ public var xMax:int:对象 blit 位置最大的 x 值
- \_ public var yMax:int:对象 blit 位置最大的 y 值
- \_ public var xMin:int:对象 blit 位置最小的 x 值
- \_ public var yMin:int: 对象 blit 位置最小的 y 值

这些公共函数 :

- \_ public function BasicBlitArrayObject(xMin:int,xMax:int, yMin:int, yMax:int ):

构造函数根据参数为对象设置最大和最小的 blit 位置(x 和 y)。根据这个基类的子类来决定更新环节, 怎么处理属性值。

- \_ public function updateFrame(inc:int):void :

updateFrame 函数传入一个增量, 设置对象的 frame 属性。然后将 bitmapData 的引用赋值给一个 animationList[frame]项。

- \_ public function render(canvasBitmapData:BitmapData):void:

渲染函数持有一个单个的 BitmapData 实例, canvasBitmapData。copyPixels 操作上, 复制给 canvasBitmapData 的 bitmapData。

这里有 BlitArrayObject 类的完整源代码 :

```
package com.efg.framework
{
import flash.display.BitmapData;
import flash.geom.Point;
import flash.events.EventDispatcher;
```



```
import flash.geom.Rectangle;
public class BasicBlitArrayObject extends EventDispatcher{
    public var x:Number = 0;
    public var y:Number = 0;
    public var nextX:Number = 0;
    public var nextY:Number = 0;
    public var dx:Number = 0;
    public var dy:Number = 0;
    public var frame:int = 0;
    public var bitmapData:BitmapData;
    public var animationList:Array=[];
    public var point:Point = new Point(0, 0);
    public var speed:Number=0;
    public var xMax:int=0;
    public var yMax:int=0;
    public var xMin:int=0;
    public var yMin:int=0;
    public function BasicBlitArrayObject(xMin:int,xMax:int, yMin:int, yMax:int ) {
        this.xMin = xMin;
        this.xMax = xMax;
        this.yMin = yMin;
        this.yMax = yMax;
    }
    public function updateFrame(inc:int):void {
        frame += inc;
        if (frame > animationList.length-1) {
            frame = 0;
        }
        bitmapData = animationList[frame];
    }
    public function render(canvasBitmapData:BitmapData):void {
        x = nextX;
        y = nextY;
        point.x = x;
        point.y = y;
        canvasBitmapData.copyPixels(bitmapData, bitmapData.rect, point);
    }
    public function dispose():void {
        bitmapData.dispose();
        bitmapData = null;
    }
}
```



```
animationList = null;
point = null;
}
}
}
```

接下来的两个类是 BasicBlitArrayObject 的子类，BasicBlitArrayParticle 和 BasicBlitArrayProjectile。他们足够普遍，能放入 framework，而不是 Blaster Mines 包。

## 设计 BasicBlitArrayParticle 类

BasicBlitArrayParticle 类是 BlitArrayObject 的子类，将会使用在我们游戏的基本爆炸粒子。这个类将会被添加到框架包里，位置在 com.efg.framework

这里有一些公共的属性。

\_ public var lifeDelayCount:int:计算动画改编里的帧数

\_ public var lifeDelay:int: lifeDelayCount 计算的帧数。

接下来是私有属性：

\_ private var remove:Boolean:如果粒子需要从屏幕上移除，则为真。

并且这些是公开属性：

\_ public function BasicBlitArrayParticle(xMin:int,xMax:int, yMin:int, yMax:int

): BasicBlitArrayObject 的全部子类的构造函数，传入的参数和父类一样，使用 super(xMin,xMax,yMin,yMax)调用。

public function update(timeBasedModifier:Number=1 ):Boolean:

更新函数的作用是，将 dx 和 dy 值（同 speed 一起）赋值给对象的 nextX 和 nextY 属性。timeBasedModifier 乘以 nextX 和 nextY 值，这个值用来保存对象在一秒内对象移动距离的常数，不论游戏 SWF 的运行帧率设置为多少。这个概念已经在“Creating the time-based step timer”这一章节里描述过了。

```
nextX+=dx*speed*timeBasedModifier;
nextY+=dy*speed*timeBasedModifier;
```

这回更新 lifeDelayCount 变量和改变对象的 bitmapData。animationList 数组的长度作为粒子的生命期使用。如果当前对象帧率比 animationList 长度大，remove 就设置为 true。同样的，如果粒子的 nextX 或者 nextY 值在设置的边界值，xMax,yMax 和 xMin,yMin 以外，remove 也设置为 true。

函数将 remove 值作为返回值，返回给 caller

这里是这个类的完整代码：

```
package com.efg.framework
```



```
{
import com.efg.framework.BasicBlitArrayObject;
public class BasicBiltArrayParticle extends BasicBlitArrayObject{
public var lifeDelayCount:int = 0;
public var lifeDelay:int=0;
private var remove:Boolean = false;
public function BasicBiltArrayParticle(xMin:int,xMax:int, yMin:int, yMax:int ) {
super(xMin, xMax, yMin, yMax);
}
public function update(step:Number=1 ):Boolean {
remove = false;
nextX+=dx*speed*step;
nextY+=dy*speed*step;
if (lifeDelayCount > lifeDelay) {
lifeDelayCount = 0;
frame++;
if (frame == animationList.length) {
remove = true;
}else {
bitmapData = animationList[frame];
}
}else {
lifeDelayCount++;
}
if (nextX > xMax || nextX < xMin || nextY > yMax || nextY < yMin) {
remove = true;
}
return(remove);
}
}
}
```

## 设计 BasicBlitArrayProjectile 类

BasicBlitArrayProjectile 类也是 BasicBlitArrayObject 的子类。它是用来描述游戏里玩家飞船发射出来的炮射导弹的。这个类将添加到游戏的框架包，位于 com.efg.framework;

这个类没有额外的公共属性，但是又一些公共函数：

构造函数跟 BasicBlitArrayParticle 的版本一样。

```
_ public function update(xAdjust:Number, yAdjust:Number,timeBasedModifier:
Number=1 ):Boolean {
```



update 函数接受 xAdjust 和 yAdjust 值。这两个值是用来修改炮弹的速度(如果需要的话)并且将玩家的速度添加到炮弹。这样你永远不会看见炮弹会比发射他们的对象慢。nextX 和 nextY 值将会乘以 timeBasedModifier。这个值是用来保存对象单秒内移动的距离,无论游戏 SWF 运行的帧率是多少。这个已经在“创建基于时间的步进计时器”这一章里讨论过了。

算法是这样的：

```
nextx+=(dx*(speed+Math.abs(xAdjust))) *timeBasedModifier;
nexty+=(dy*(speed+Math.abs(yAdjust))) *timeBasedModifier;
```

为了决定是否从屏幕上移除炮弹,新的 nextX 和 nextY 属性和这个对象的最大最小的, x 和 y 值进行匹配,如果对象在这些边界之外,就返回 true 给调用者。

```
if (nextX > xMax || nextX < xMin || nextY > yMax || nextY < yMin) {
    remove = true;
}
return(remove);
```

这里是类的所有源码：

```
package com.efg.framework
{
    import com.efg.framework.BasicBlitArrayObject;
    import com.efg.framework.BasicBiltArrayProjectile;
    public class BasicBiltArrayProjectile extends BasicBlitArrayObject{
        public function BasicBiltArrayProjectile(xMin:int,xMax:int, yMin:int, yMax:int ) {
            super(xMin, xMax, yMin, yMax);
        }
        public function update(xAdjust:Number, yAdjust:Number,step:Number=1 ):Boolean {
            //x adjust and y adjust change speed of projectile.
            //in this case they are used to ensure that sprojetciles are as fast
            //add xMove to dx and dy so the missiles fire faster if ship is moving faster
            nextX+=(dx*(speed+Math.abs(xAdjust)))*step;
            nextY+=(dy*(speed+Math.abs(yAdjust)))*step;
            if (nextX > xMax || nextX < xMin || nextY > yMax || nextY < yMin) {
                return(true)
            }else {
                return(false);
            }
        }
    }
}
```

## 设计 BlitArrayPlayerFollowMouse 类



BlitArrayPlayerFollowMouse 类是 BasicBlitArrayObject 最复杂的一个子类。它包含了一些 Blaster Mines 游戏的特别代码，所以他应该在 Blaster Mines 游戏包里而不是框架包里。

BlitArrayPlayerFollowMouse 基于一个 update 函数，将一个计算好的偏移值传递给该函数，来让玩家以正确的速度和方向跟着鼠标移动。当前的鼠标位置会和一个 delay 值一起传递给 update 函数，来减缓玩家飞船的移动速度，这样它就不会将他直接粘在鼠标指针上移动了。这样飞船的移动更逼真一些。

使用鼠标控制和允许自动导弹自动开火是相对来说比较新的和基于 Flash 的控制机制。这样在那些兼容 Flash Player 游戏程序的手持式触摸屏终端上也能很好的瞄准。

这个包位于 com.efg.games.blastermines.

这里有些公共属性：

\_ public var xMove:int:基于鼠标位置，x 方向上计算出来的新变化。

\_ public var yMove:int:基于鼠标位置，y 方向上计算出来的新变化。

\_ public var shipBitmapData:BitmapData:持有玩家飞船的 BitmapData 版本的 drawingCanvas。

\_ public var shieldBitmapData:BitmapData: 持有 BitmapData 版本的飞船护盾。

\_ public var shieldRender:Boolean:如果玩家的飞船被 Mine 实例撞上了，告诉渲染函数来显示玩家飞船周围的护盾。

\_ public var shieldCount:int:在设置 shieldRender 为 false 之前，从 0 开始计数到 shieldLife 帧运行的时候，这样护盾就能以 shieldLife 值保持运行。

\_ public var shieldLife:int:当玩家被一个 Mine 撞击后，shield 被激活的帧数。

这里是私有属性：

\_ private var xChange:int: mouseY 值和对象的 Y 值之间的不同。

\_ private var yChange:int: mouseX 值和对象的 x 值之间的不同。

\_ private var radians:Number:保存玩家当前旋转的角度，以弧度为单位。

\_ private var degrees:int:保存玩家当前旋转的角度。

\_ private var drawingCanvas:Shape:在 shipBitmapData 变量里绘制飞船的外观和护盾之前，构造他们。

这里有 3 个公共函数。

第一个，和 BasicBlitArrayParticle 一样的版本。

这里是第二个函数：

```
public function update(mousePositionX:Number, mousePositionY:Number,
delay:int,timeBasedModifier:Number=1):void
```



这个函数用当前的鼠标 x 和 y 值作为一个延迟值。和前面提到的一样，延迟用来控制玩家飞船的速度。timeBaseModifier 将用来保存对象每秒移动在一个固定的速率上，无论游戏运行在什么样的帧率下。

基于传进去的值，计算新的角度和移动改变值。这里是算法：

我们计算飞船新角度的弧度值，使用鼠标 x 和 y 位置与玩家 x 和 y 位置之间的不同，这个计算使用 Math.atan2 函数。

接下来，我们先计算出弧度值，然后用弧度值计算出角度值。

```
radians = Math.atan2((mousePositionY)-y,(mousePositionX)-x);
degrees= (radians * (180 / Math.PI));
```

然后，我们计算出鼠标和玩家之间不同的 x 和 y 值。

```
yMove=(yChange/delay)*timeBasedModifier;
xMove=(xChange/delay)*timeBasedModifier;
```

xMove 和 yMove 值乘以 timeBasedModifier.这个计算保证无论游戏 SWF 运行的帧率是多少，对象在每秒钟移动的距离。

接下来，我们用不同的变化除以一个 delay，这样飞船就不会粘着鼠标了。

```
yMove=yChange/delay;
xMove=xChange/delay;
```

nextX 和 nextY 值分别的加上 xMove 和 yMove 并且和 minimum 和 maximum x 和 y 值进去匹配。在这个游戏里，屏幕相对的边没有歪曲（不像 Atari Asteroids），所以如果他尝试移出这些边界，玩家的飞船对象就会停止。

```
nextX+=yMove;
nextY+=xMove;
if (nextY > xMax) {
nextX = xMax;
}else if (nextX < +xMin) {
nextX = xMin;
}
if (nextY > yMax) {
nextY = yMax;
}else if (nextY < yMin) {
nextY = yMin;
}
```

最后 基于对象的新的角度在 animationList 数组里选中的新的一帧动画（一个旋转过的 Bitmap 实例）。如果这个角度值小于 0，我们需要的偏移量为 359,以便能呈现出动画帧数的适当的数组索引。

```
frame = degrees;
```



```
if (degrees < 0) {
    frame = 359+degrees;
}
bitmapData=animationList[frame];
```

BlitArrayPlayerFollowMouse 的第三个函数：

```
public function createPlayerShip(spriteGlowFilter:GlowFilter):void
```

这个函数将会以矢量图形绘制飞船，然后使用 BlitArrayAsset 类来创建一个 360 度旋转的飞船的 BitmapData 对象。

```
shipBitmapData.draw(drawingCanvas);
shipBitmapData.applyFilter(shipBitmapData, shipBitmapData.rect, new Point(0,0), _
spriteGlowFilter);
animationList=tempBlitArrayAsset.createRotationBlitArrayFromBD(shipBitmapData, 1,90);
```

一旦这个飞船被绘制进了 drawingCanvas，我们就是用 BitmapData.draw 方法来讲它复制到 shipBitmapData。我们会在“添加 Blaster Mines 游戏的初始化函数”这一章节里实践。

这里是这个类的完整源码：

```
package com.efg.games.blastermines
{
    import com.efg.framework.BasicBlitArrayObject;
    import com.efg.framework.BlitArrayAsset;
    import flash.display.BitmapData
    import flash.display.Shape;
    import flash.geom.Point;
    import flash.filters.GlowFilter;
    public class BlitArrayPlayerFollowMouse extends BasicBlitArrayObject{
        public var xMove:int;
        public var yMove:int;
        private var xChange:int;
        private var yChange:int;
        private var radians:Number;
        private var degrees:int;
        private var drawingCanvas:Shape = new Shape();
        public var shipBitmapData:BitmapData = new BitmapData(32, 32, true, 0x00000000);
        public var shieldBitmapData:BitmapData = new BitmapData(32, 32, true, 0x00000000);
        public var shieldRender:Boolean = false;
        public var shieldCount:int = 0;
        public var shieldLife:int = 5;
        public function BlitArrayPlayerFollowMouse(xMin:int,xMax:int, yMin:int, yMax:int ) {
```



```
super(xMin, xMax, yMin, yMax);
}
public function update(mousePositionX:Number, mousePositionY:Number, _
delay:int,step:Number=1):void {
radians = Math.atan2((mousePositionY)-y,(mousePositionX)-x);
degrees= (radians * (180 / Math.PI));
yChange= (mousePositionY-y);
xChange= (mousePositionX-x);
yMove=(yChange/delay)*step;
xMove=(xChange/delay)*step;
nextY+=yMove;
nextX+=xMove;
if (nextX > xMax) {
nextX = xMax;
}else if (nextX < +xMin) {
nextX = xMin;
}
if (nextY > yMax) {
nextY = yMax;
}else if (nextY < yMin) {
nextY = yMin;
}
frame = degrees;
if (degrees < 0) {
frame = 359+degrees;
}
bitmapData=animationList[frame]
}
public function createPlayerShip(spriteGlowFilter:GlowFilter):void {
var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();
drawingCanvas.graphics.clear();
drawingCanvas.graphics.lineStyle(1, 0xffffffff);
drawingCanvas.graphics.moveTo(15, 7);
drawingCanvas.graphics.lineTo(7, 24);
drawingCanvas.graphics.lineTo(15, 19);
drawingCanvas.graphics.moveTo(16, 19);
drawingCanvas.graphics.lineTo(24, 24);
drawingCanvas.graphics.lineTo(16, 7);
trace("drawingCanvas.height=" + drawingCanvas.height);
trace("drawingCanvas.width=" + drawingCanvas.width);
```



```
shipBitmapData.draw(drawingCanvas);
shipBitmapData.applyFilter(shipBitmapData, shipBitmapData.rect, new Point(0,0),_
spriteGlowFilter);
animationList=tempBlitArrayAsset.createRotationBlitArrayFromBD_
(shipBitmapData, 1,90);
/** end player ship
/** shield
drawingCanvas.graphics.clear();
drawingCanvas.graphics.lineStyle(3, 0xffffffff);
drawingCanvas.graphics.drawCircle(15, 15, 14);
shieldBitmapData.draw(drawingCanvas);
/** end shield
}
}
}
```

就像我们逐步的浏览这个游戏的代码，我们会更详细的讨论这个类。接下来，我们会为我们的游戏创建三个管理类，将会将一组对象的功能压缩要一起。第一个类是 MineManager。这个类将会管理玩家必须摧毁的 Mine 敌人飞船

因为这些类是 Blaster Mines 游戏的核心，当我们阅读游戏代码的时候我们会自信的讨论的。到此为止，让我们简要的看一下每个的代码和每个所包含的一个公共和稀有属性还有函数的简短的描述。

## 设计 MineManager 类

MineManager 类持有玩家必须摧毁的，活动 Mine 实例的列表。在本节我们接下来的讨论中，Mines 就是 Mine 类的实例。这个类将是 Blaster Mines 游戏包：com.efg.games.blastermines。

这里是公共属性：

\_ public var mineBitmapData:BitmapData: 只是一个 Bitmap 用来在 drawingCanvas 里构建 mine 的外观后呈现它。

\_ public var mineAnimationFrames:Array: 用来存放 360 帧旋转的 BitmapData 对象，表现 Mine 的旋转。

\_ public var mines:Array: 这里讲保存所有处于活动状态的 Mine 类实例。

\_ public var tempMine:Mine: 这个是一个类共用变量，用在 tempMine 所需要的所有操作里。

\_ public var mineCount:int: 这个持有 mines 数组的当前长度，这样就不用再每个循环迭代内部在重新计算了。

这些是私有变量：

\_ private var drawingCanvas:Shape: 在矢量 mine 外观被绘制到 mineBitmapData 上之前，构造它的容器。



\_ private var point0:Point: 0x 和 0y 的公共点实例 能用于所有 mineBitmapData 的绘制和滤镜操作。

这里是公共函数：

\_ public function MineManager:构造函数不做任何处理。

\_ public function createLevelMines(spriteGlowFilter:GlowFilter,level:int,

levelColor:uint):void:这个函数创建了当前游戏级别所有的 Mine 实例，并且订制了 mine 的外观，把它绘制到 drawingCanvas 里。它使用 BlitArrayAsset 类来创建一个存放着 360 个方向的独立的 BitmapData 实例的数组。

当我们浏览 BlasterMines.as 游戏时，我们深入这个类的细节部分。通过 BlasterMines 类，我们能更有感觉的了解他是怎么工作的。

这里是这个类的全部代码：

```
package com.efg.games.blastermines
{
import flash.display.BitmapData;
import flash.filters.GlowFilter;
import flash.display.Shape;
import flash.geom.Point;
import com.efg.framework.BlitArrayAsset;
public class MineManager {
public var mineBitmapData:BitmapData ;
public var mineAnimationFrames:Array = [];
public var mines:Array;
public var tempMine:Mine;
public var mineCount:int;
private var drawingCanvas:Shape = new Shape();
private var point0:Point = new Point(0, 0);
public function MineManager() {}
public function createLevelMines(spriteGlowFilter:GlowFilter,level:int, _
levelColor:uint):void {
/** Mines look
mineBitmapData= new BitmapData(32, 32, true, 0x00000000);
var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();
drawingCanvas.graphics.clear();
drawingCanvas.graphics.lineStyle(2, 0xffffffff);
drawingCanvas.graphics.moveTo(6, 6);
drawingCanvas.graphics.lineTo(25, 6);
drawingCanvas.graphics.lineTo(25, 22);
drawingCanvas.graphics.lineTo(6, 22);
```



```
drawingCanvas.graphics.lineTo(6, 6);
drawingCanvas.graphics.moveTo(18, 8);
drawingCanvas.graphics.lineTo(18, 16);
drawingCanvas.graphics.lineTo(12, 16);
drawingCanvas.graphics.lineTo(12, 12);
drawingCanvas.graphics.moveTo(9, 23);
drawingCanvas.graphics.lineTo(9, 25);
drawingCanvas.graphics.moveTo(15, 23);
drawingCanvas.graphics.lineTo(15, 25);
drawingCanvas.graphics.moveTo(21, 23);
drawingCanvas.graphics.lineTo(21, 25);
spriteGlowFilter.color = levelColor;
mineBitmapData.draw(drawingCanvas);
mineBitmapData.applyFilter(mineBitmapData, mineBitmapData.rect, point0, _
spriteGlowFilter);
tempBlitArrayAsset = new BlitArrayAsset();
mineAnimationFrames=tempBlitArrayAsset.createRotationBlitArrayFromBD_
(mineBitmapData, 1,90);
/** end of mines look
/** create mines for level
mines = [];
for (var ctr:int=0;ctr<30+20*level;ctr++) {
var tempMine:Mine=new Mine(5,765,5,765);
tempMine.dx=Math.cos(6.28*((Math.random()*360)-90)/360.0);
tempMine.dy = Math.sin(6.28 * ((Math.random() * 360) - 90) / 360.0);
if (level % 2 == 0) {
tempMine.y = 700
}else {
tempMine.y = 100
}
tempMine.x = 100;
tempMine.nextX = tempMine.x;
tempMine.nextY = tempMine.y;
tempMine.frame = int((Math.random() * 359)) ;
tempMine.animationList = mineAnimationFrames;
tempMine.bitmapData = tempMine.animationList[tempMine.frame];
tempMine.speed = (Math.random()*1)+2+level;
mines.push(tempMine);
}
}
```



```
}  
}
```

## 设计 ParticleManager 类

ParticleManager 类将会包含 Mine 实例爆炸的粒子的设计。它会创建一个内存优化的对象池。我们会在这里展现一些类的基本描述，然后我们会继续进入 BlasterMines.as 类，进行一些细节的讨论。Particles 将会是 BasicBlitArrayParticle 类的实例。每一级的外观都是独特的，并且颜色和玩家必须摧毁的 Mine 实例一致。

包的位置是 com.efg.games.blastermine.

接下来是这个类的公共属性：

\_ public var particleBitmapData:BitmapData: 这个是粒子的 BitmapData 外观。每一个级别都有和 Mines 样色相符的独特的颜色。

\_ public var particleAnimationFrames: 当他们的生命终结的时候，粒子就会消失。消失的动画帧将会存放在这个数组里。将会用 BlitArrayAsset 类创建。

\_ public var particles:Array:包含当前活动粒子的数组。

\_ public var particlePool:一个用作粒子缓冲池的数组

\_ public var particleCount: 这是个类级别变量，用来保存粒子数组的长度。

\_ public var particlePoolCount: 这是个类级别变量，用来保存 particlePool 数组的长度。

\_ public var tempParticle:BasicBlitArrayParticle: 可重用的类级别粒子实例，用于粒子数组的循环。

\_ public var particlePoolMax: 一个单独的时间里，存放屏幕上所有的粒子实例的数量的属性。

\_ public var particlesPerExplode: 在每个爆炸中粒子实例的数量。

这些是私有属性：

\_ private var drawingCanvas:Shape: 创建粒子的外观的容器。

\_ private var point0:Point:绘制操作中可重用的 Point 实例。

下面是类的公共函数：

\_ public function ParticleManager():构造函数是一个空函数。

\_ public function createParticlePool(maxParticles:int):void: BlasterMines.as 类将会在每个新等级前调用这个函数，来创建粒子池。在我们讨论 BlasterMines.as 代码的时候，我们会具体的了解并且会更详细的讨论它。

\_ public function createLevelParticles(spriteGlowFilter:GlowFilter,levelColor:uint):void: 这个函数在每个新级别之前被 BlasterMines.as 调用。用来创建跟 Mine 实例颜色一致的粒子外观。



这里是这个类的完整代码：

```
package com.efg.games.blastermine
{
import flash.display.BitmapData;
import com.efg.framework.BlitArrayAsset;
import com.efg.framework.BasicBiltArrayParticle;
import flash.display.BitmapData;
import flash.display.Shape;
import flash.geom.Point;
import flash.filters.GlowFilter;
public class ParticleManager {
public var particleBitmapData:BitmapData;
public var particleAnimationFrames:Array = [];
public var particles:Array = [];
public var particlePool:Array = [];
public var particleCount:int=0;
public var particlePoolCount:int=0;
public var tempParticle:BasicBiltArrayParticle
public var particlePoolMax:int = 500;
public var particlesPerExplode:int = particlePoolMax / 20;
private var drawingCanvas:Shape = new Shape();
private var point0:Point = new Point(0, 0);
public function ParticleManager() {}
public function createParticlePool(maxParticles:int):void {
particlePool = [];
particles = [];
for (var particleCtr:int=0;particleCtr<maxParticles;particleCtr++) {
var tempParticle:BasicBiltArrayParticle = new BasicBiltArrayParticle_
(0,799,0,799);
particlePool.push(tempParticle);
}
}
public function createLevelParticles(spriteGlowFilter:GlowFilter, _
levelColor:uint):void {
var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();
particleBitmapData = new BitmapData(8, 8, true, 0x00000000);
drawingCanvas.graphics.clear();
drawingCanvas.graphics.lineStyle(1, 0xffffffff);
drawingCanvas.graphics.drawRect(3, 3, 2, 2);
particleBitmapData.draw(drawingCanvas);
}
```



```
spriteGlowFilter.color = levelColor;
particleBitmapData.applyFilter(particleBitmapData, particleBitmapData.rect, _
point0, spriteGlowFilter);
tempBlitArrayAsset= new BlitArrayAsset();
tempBlitArrayAsset.createFadeOutBlitArrayFromBD(particleBitmapData, 30);
particleAnimationFrames = tempBlitArrayAsset.tileList;
}
}
}
```

## 设计 ProjectileManager 类

ProjectileManager 类的结构几乎跟 ParticalManager 类一致。主要的不同在于 projectilePool 每个游戏 session 只创建一次，不像粒子池每个级别都重新创建。projectiles 是 BasicBlitArrayProjectile 类的实例。

这个包的位置在 com.efg.games.blastermine.

这个类的公共属性是：

\_ public var projectileBitmapData:BitmapData:这个是一个 Projectiles 的 BitmapData 外观。

\_ public var projectileAnimationFrames: Projectile 飞过空气时旋转。旋转的动画帧将会存放在这个数组里。将会用 BlitArrayAsset 类一起创建。

\_ public var projectiles:Array: 这个属性是一个当前活动活动的 Projectile 的数组。

\_ public var projectilePool:这个保存了 projectile 池的数组。

\_ public var projectile Count:类级别的变量来存放 projectiles 数组的长度。

\_ public var projectile PoolCount: 类级别的变量来保存 projectilePool 数组的长度。

\_ public var tempProjectile:BasicBiltArrayProjectile: 可重用的类级别 Projectile 实例 在 Projectile 数组的循环中使用。

\_ public var projectilePoolMax: 这属性存放了在一个单个时间里当前屏幕里 projectiles 的总数量。

私有属性是

\_ private var drawingCanvas:Shape:绘制 projectile 外观的容器。

\_ private var point0:Point:绘制操作中可重用的点实例。

这些公共函数：

\_ public function Projectile Manager(): 构造函数是一个空函数。



\_ public function createProjectilePool(maxParticles:int):void: BlasterMines.as 类在他的 init 函数之前调用这个函数，来创建 projectile 池。在我们讨论 BlasterMines.as 代码的时候会具体了解和更多的讨论它。

\_ public function createProjectilePool(maxProjectiles:int):void: BlasterMines.as 在 init 之前调用这个函数，它创建 projectile 的外观。

这里是这个类的完整代码：

```
package com.efg.games.blastermines
{
import flash.display.BitmapData;
import com.efg.framework.BlitArrayAsset;
import com.efg.framework.BasicBiltArrayProjectile;
import flash.display.BitmapData;
import flash.display.Shape;
import flash.geom.Point;
import flash.filters.GlowFilter;
public class ProjectileManager {
public var projectileBitmapData:BitmapData = new BitmapData(8, 8, true, 0x00000000);
public var projectileAnimationFrames:Array = [];
public var projectiles:Array=[];
public var lastProjectileShot:Number = 0;
public var projectileCount:int;
public var tempProjectile:BasicBiltArrayProjectile;
public var projectilePool:Array = [];
public var projectilePoolCount:int;
private var drawingCanvas:Shape = new Shape();
private var point0:Point = new Point(0, 0);
public function ProjectileManager() {}
public function createProjectiles(spriteGlowFilter:GlowFilter):void {
//projectile look
var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();
drawingCanvas.graphics.clear();
drawingCanvas.graphics.lineStyle(1, 0xffffffff);
drawingCanvas.graphics.drawRect(3, 3, 2, 2);
projectileBitmapData.draw(drawingCanvas);
projectileBitmapData.applyFilter(projectileBitmapData, _
projectileBitmapData.rect, point0, spriteGlowFilter);
tempBlitArrayAsset= new BlitArrayAsset();
projectileAnimationFrames=tempBlitArrayAsset.createRotationBlitArrayFromBD_
(projectileBitmapData, 10,0);
```



```
}  
public function createProjectilePool(maxProjectiles:int):void {  
    for (var projectileCtr:int = 0; projectileCtr < maxProjectiles; projectileCtr++) {  
        var tempProjectile:BasicBltArrayProjectile=new BasicBltArrayProjectile_  
            (0,799,0,799);  
        tempProjectile.animationList=projectileAnimationFrames;  
        tempProjectile.bitmapData = tempProjectile.animationList[0];  
        projectilePool.push(tempProjectile);  
    }  
}  
}  
}
```

ProjectileManager 类是最后一个管理类。现在我们将看一看我们游戏最后一个自定义类—Mine 类。

## 设计 Mine 类

Mine 类是 BasicBltArrayObject 类的扩展,定义的更新函数用了基于时间的步进计时器实现了步进修改器。这个类很简单,我们只是展示一下这个代码,到我们讲到 BlasterMines 更新函数的时候再来讨论这个更新函数。

包的位置在:com.efg.games.blastermines.

这里是完整的类代码:

```
package com.efg.games.blastermines  
{  
    import com.efg.framework.BasicBltArrayObject;  
    public class Mine extends BasicBltArrayObject{  
        public function Mine(xMin:int,xMax:int, yMin:int, yMax:int) {  
            super(xMin, xMax, yMin, yMax);  
        }  
        public function update(step:Number=1):void {  
            nextX+=dx*speed*step;  
            nextY+=dy*speed*step;  
            //bounce  
            if (nextX > xMax) {  
                nextX = xMax;  
                dx *= -1;  
            }else if (nextX < xMin) {  
                nextX = xMin;  
                dx *= -1;  
            }  
        }  
    }  
}
```



```

}
if (nextY > yMax) {
    nextY = yMax;
    dy *= -1;
}else if (nextY < yMin) {
    nextY = yMin;
    dy *= -1;
}
}
}
}
}

```

这个类里最重要需要注意的东西是 Mine 在撞击到世界边缘后四处反弹的代码。当一个 Mine 撞击到世界的边界，从它来的方向的相反角度发射出去。我们只在 Mine 撞击的世界边界轴这样做。这样，他就不会简单的弹回而是反射会到世界里。

那就是 Mine 类。现在，我们看一下 BlasterMines Game 类子类和我们创建游戏所需要实现的所有逻辑。作为 BlasterMines.as 类的技术设计，我们将会在本节里反复的阅读代码，并且描述至今为止还未被讨论的函数的新部分。你会发现我们已经讲了大部分的 BlasterMines.as。

## 创建 Blaster Mines 类

我们将会创建 BlasterMines.as 类。前面一章我们不停的重复一个工作的游戏文件，这个工作流和前面一章一样。在这一章，我们会快速的略过一大段你前面已经看过的代码，只会指出一些前面已经讨论过的理论和任何没有讨论过的新的代码和理论的实现。因为我们已经在这章里讲了一大堆一大堆高级主题，如果你没有阅读过前面的章节，你想自己实现一部分，再直接尝试优化游戏代码之前。

让我们来创建这个类做为 Blaster Mines 游戏包的一个新文件。这里是类文件在 Flex SDK 的名字和位置(用 Flash Develop)：

```
/source/projects/blastermines/flexIDE/com/efg/games/blastermines/BlasterMines.as
```

这里是 Flex SDK ( 使用 Flash Develop )

```
/source/projects/blastermines/flexSDK/src/com/efg/games/blastermines/BlasterMines.as
```

## 创建 Blaster Mines 类壳

壳包含了 BlasterMines.as 类文件的类引用的部分，变量定义部分，和构造函数。

```

package com.efg.games.blastermines
{
    import com.efg.framework.Game;
    import com.efg.framework.GameFrameWork;
    import com.efg.framework.BasicBiltArrayParticle;
    import com.efg.framework.BasicBiltArrayProjectile;
}

```



```
import flash.display.*;
import flash.events.*;
import flash.geom.*;
import flash.filters.*;
import com.efg.framework.BasicBlitArrayObject;
import com.efg.framework.BlitArrayAsset;
import com.efg.framework.CustomEventLevelScreenUpdate;
import com.efg.framework.CustomEventScoreBoardUpdate;
import com.efg.framework.CustomEventSound;
/**
 * ...
 * @author Jeff Fulton
 */
public class BlasterMines extends Game{
    public static const GAME_OVER:String = "game over";
    public static const NEW_LEVEL:String = "new level";
    public static const STATE_SYSTEM_GAME_PLAY:int = 0;
    public static const STATE_SYSTEM_PLAYER_EXPLODE:int = 1;
    private var systemFunction:Function;
    private var currentSystemState:int;
    private var nextSystemState:int;
    private var lastSystemState:int;
    /** game loop
    private var gameOver:Boolean = false;
    //leveldata
    private var levelColors:Array = [NaN, 0xff0000, 0x00ff00, 0x0000ff, 0xffff00, _
    0x00ffff, 0xffaa00, 0xaa00ff, 0x00ffaa, 0x00aaff];
    private var levelColor:uint;
    private var level:int = 0;
    private var score:int = 0;
    private var shield:int = 10;
    private var maxLevel:int = levelColors.length;
    private var playerStarted:Boolean = false;
    private var playerExplosionParticles:Array = [];
    //Canvas and background
    private var backgroundBitmapData:BitmapData = new BitmapData(800, 800, false, _
    0x0000000);
    private var canvasBitmapData:BitmapData = new BitmapData(800, 800, false, 0x0000000);
    private var canvasBitmap:Bitmap = new Bitmap(canvasBitmapData);
    private var canvasRect:Rectangle = new Rectangle(0,0,400,400);
```



```
//drawing
private var drawingCanvas:Shape = new Shape();
//player
private var player:BlitArrayPlayerFollowMouse = new BlitArrayPlayerFollowMouse_
(1, 767, 1, 767);
//mineManager
private var mineManager:MineManager = new MineManager();
public var tempMine:Mine;
//projectileManagere
private var projectileManager:ProjectileManager = new ProjectileManager();
private var tempProjectile:BasicBiltArrayProjectile;
//particleManager.particles
private var particleManager:ParticleManager = new ParticleManager();
private var tempParticle:com.efg.framework.BasicBiltArrayParticle;
//reused rectangles/points
private var rect32:Rectangle = new Rectangle(0, 0, 32, 32);
private var point0:Point = new Point(0, 0);
private var spriteGlowFilter:GlowFilter = new GlowFilter(0x0066ff, 1, 3, 3, 3, _
3, false, false);
private var canvasBitmapGlowFilter:GlowFilter=new GlowFilter(0x0066ff, .5, 400, _
400, 1, 1, true, false);
//scoreBoard objects - for optimization
private var customScoreBoardEventScore:CustomEventScoreBoardUpdate = new _
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,_
Main.SCORE_BOARD_SCORE, "");
private var customScoreBoardEventShield:CustomEventScoreBoardUpdate = new _
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT, _
Main.SCORE_BOARD_SHIELD, "");
private var customScoreBoardEventLevel:CustomEventScoreBoardUpdate= new _
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,_
Main.SCORE_BOARD_LEVEL, "");
private var customScoreBoardEventParticlePool:CustomEventScoreBoardUpdate = new _
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,_
Main.SCORE_BOARD_PARTICLE_POOL,"");
private var customScoreBoardEventParticleActive:CustomEventScoreBoardUpdate = new _
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,_
Main.SCORE_BOARD_PARTICLE_ACTIVE,"");
private var customScoreBoardEventProjectilePool:CustomEventScoreBoardUpdate = _
new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,_
Main.SCORE_BOARD_PROJECTILE_POOL, "");
```



```
private var customScoreBoardEventProjectileActive:CustomEventScoreBoardUpdate = _
new CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,_
Main.SCORE_BOARD_PROJECTILE_ACTIVE, "");
//math look up tables
//private var rotationVectorList:Array = [];
private var rotationVectorList:Vector.<Point>=new Vector.<Point>(360,false);
//radar
private var radarBitmap:Bitmap = new Bitmap(canvasBitmapData);
public function BlasterMines() {
trace("constructor");
init();
}
}
}
```

变量声明部分和你至今为止在本书里看到的其他部分十分相似。我们将他分割成小段以便更容易理解。我们在本章节之前的章节已经讨论过很多很多了。当我们继续进行 BlasterMines.as 类的剩下部分代码，我们会仔细讨论新的概念并且简要的讲一下我们这章或者是前面的章节没讲到的部分。

## 添加 Blaster Mines 游戏的 init 函数

BlasterMines.as 的 init 函数设置所有将运行的游戏的类。他包含了渲染分析器的设定代码，也包含了创建 ship 资源，炮弹资源和炮弹对象池的代码。

首先，我们看一下 setRendering 函数。我们在本章的早些时候简略的讨论一点。

```
override public function setRendering(profiledRate:int, framerate:int):void {
var percent:Number=profiledRate / framerate
trace("framepercent=" + percent);
trace("stage=" + stage);
if (percent>=.85) {
frameRateMultiplier = 2;
}
trace("frameRateMultiplier=" + frameRateMultiplier);
}
```

setRendering 函数是公共的，在 Main.as 类中调用。从 FrameRateProfiler 类中传入 profiledRate，同时也是游戏的设计 frameRate。如果 profiledRate 不小于设计 frameRate 的 85%，frameRateMultiplier 变量就设置为 2。这个设定允许我们加倍爆炸时粒子的数量。我们也将粒子池的大小翻倍了。这个函数只是作为一个小的例子，你如何使用分析得出的信息来为不同的运行电脑创建一个独特的体验。

我们还不能访问舞台变量，因为当 init 函数被调用的时候，BlasterMines.as 游戏还没有被添加到 Main.as 舞台显示列表。当 newGame 函数被调用的时候舞台才是可用。在那个函数里，我们会基于新的 frameRateMultiplier 值来设置 stage.quality。



下一个运行的就是 init 函数，我们在这里创建对象和设置 manager 类

```
private function init():void {
    this.focusRect = false;
    //init radar
    radarBitmap.x = 420;
    radarBitmap.y = 230;
    radarBitmap.scaleX = .2;
    radarBitmap.scaleY = .2;
    createLookupTables();
    player.createPlayerShip(spriteGlowFilter);
    projectileManager.createProjectiles(spriteGlowFilter);
    projectileManager.createProjectilePool(50);
    canvasBitmap.scrollRect = new Rectangle(200,200,400,400);
    addChild(canvasBitmap);
    addChild(radarBitmap);
}
```

init 函数设置游戏的所有的初始化信息。我们前面在优化章节讨论过 radarBitmap 和 canvasBitmap.scrollRect。createLookupTables 函数也在前面讨论过，并且你会看到 createPlayerShip，createProjectiles 和 createProjectilePool 函数的详细内容，辅助管理类我们会简短的带过。有一件需要注意的是所有这些在整个游戏里只需要调用和创建一次。不要重复调用它们。

这个函数最后的任务是添加两个 Bitmap 容器到屏幕上。如果你重新调用，它们都是 canvasBitmapData 的容器。canvasBitmapData 是一个 800\*800 的用来表现整个世界的 blit 容器。canvasBitmap 使用一个 400\*400scrollRect 来对这个 800\*800 的世界进行滚屏，radarBitmap 显示一个世界改变大小的版本位于游戏屏幕的右边。

现在，我们看一看 createLookUpTables 函数：

```
private function createLookupTables():void {
    for (var ctr:int = 0; ctr < 359; ctr++) {
        var point:Point = new Point();
        point.x = Math.cos((Math.PI * ctr) / 180);
        point.y = Math.sin((Math.PI * ctr) / 180);
        rotationVectorList[ctr] = point;
    }
}
```

我们在前面已经研究过这个函数了。他的工作是创建一个叫 vectorRotationList 的 Point 对象的向量数组，存放向量移动中旋转的 360 度表现的实例。在 BlasterMines.as 初始化函数里，我们也会调用这个类的 player.createPlayerShip 方法来绘制玩家飞船到一个向量绘制容器里。玩家的对象是一个



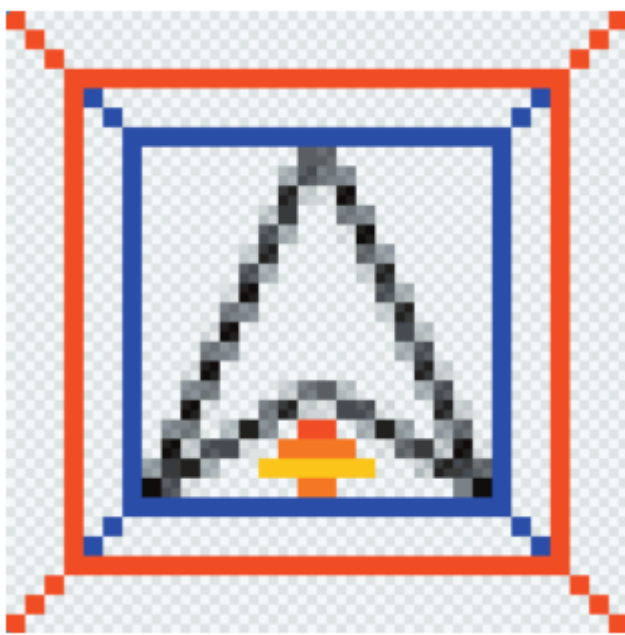
BlitArrayPlayerFollowMouse 的实例。这个容器会放置在 BitmapData 容器里 然后创建一个 BitmapData 实例的数组来存放飞船旋转 360 度的表现。

我们已经快速的掠过了这个在 BlitArrayPlayerFollowMouse 里的函数。让我们重温一遍 ,更深入的理解他的功能。

```
public function createPlayerShip(spriteGlowFilter:GlowFilter):void {
    var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();
    drawingCanvas.graphics.clear();
    drawingCanvas.graphics.lineStyle(1, 0xffffffff);
    drawingCanvas.graphics.moveTo(15, 7);
    drawingCanvas.graphics.lineTo(7, 24);
    drawingCanvas.graphics.lineTo(15, 19);
    drawingCanvas.graphics.moveTo(16, 19);
    drawingCanvas.graphics.lineTo(24, 24);
    drawingCanvas.graphics.lineTo(16, 7);
    shipBitmapData.draw(drawingCanvas);
    shipBitmapData.applyFilter(shipBitmapData, shipBitmapData.rect, new Point(0,0), _
    spriteGlowFilter);
    animationList=tempBlitArrayAsset.createRotationBlitArrayFromBD(shipBitmapData, 1,90);
    /*** end player ship
    /*** shield
    drawingCanvas.graphics.clear();
    drawingCanvas.graphics.lineStyle(3, 0xffffffff);
    drawingCanvas.graphics.drawCircle(15, 15, 14);
    shieldBitmapData.draw(drawingCanvas);
    /*** end shield
}
```

这是个有趣的函数。他的工作是以一个矢量图形的方式绘制玩家的飞船并且使用 BlitArrayAsset 类来创建一个飞船 360 度旋转的数组。每个这些都是一个独立的 BitmapData 对象 ,用作在渲染的时候 blitting 这个 canvasBitmapData。我们首先在 Adobe Fireworks 里绘制一下玩家的的飞船。

然后我们在 createPlayerShip 里 ,使用我们创建的位图图像在 drawCanvas 形状实例上将线条勾勒出来。



**Figure 11-4.** The 32 \_ 32 player ship blown up in

Fireworks design mode

在 Fireworks 里，玩家的飞船实际上是设计绘制的在 32\*32 大小的容器里，在设计视图里放大了 16 倍。蓝色的先表示发光滤镜结束的地方，红色线是旋转的缓冲地带，以便绘制的对象不会产生剪切。我们所有的游戏对象将会绘制成白色，并且全局的类级别 BlasterMines.as 的 GlowFilter 实例 spriteGlowFilter 会修改成对象的发光滤镜需要矫正的颜色。飞船的发光是蓝色的。游戏开始的时候，spriteGlowFilter 设置成蓝色，所以这里不需要修改。我为其他对象来重置它，这样在我们绘制那些东西之前需要设置这个颜色。接下来，我们在 drawCanvas 里绘制矢量线条，并且在结束的时候，我们使用 BitmapData.draw 方法来绘制对象到一个 BitmapData 容器里。我们接着用一个临时的 BlitArrayAsset 实例为玩家创建了 animationList 数组。

玩家的护盾以同样的方式创建（调用(drawCanvas.graphics.drawCircle 函数)。当玩家护盾开启的时候，就会被绘制到一个 BitmapData 实例里(shieldBitmapData)并且直接 blit 到 canvasBitmapData。

现在，我们将研究 projectileManager.createProjectiles 函数

```
public function createProjectiles(spriteGlowFilter:GlowFilter):void {
    //projectileManager.projectiles
    var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();
    drawingCanvas.graphics.clear();
    drawingCanvas.graphics.lineStyle(1, 0xffffffff);
    drawingCanvas.graphics.drawRect(3, 3, 2, 2);
    projectileBitmapData.draw(drawingCanvas);
    projectileBitmapData.applyFilter(projectileBitmapData, projectileBitmapData.rect, _
    point0, spriteGlowFilter);
    tempBlitArrayAsset= new BlitArrayAsset();
    projectileAnimationFrames=tempBlitArrayAsset.createRotationBlitArrayFromBD_
    (projectileBitmapData, 10,0);
}
```

以同样的方式将炮弹会知道玩家的飞船。这是一个简单的矩形，用着和飞船发光一致的 blue 发光滤镜(spriteGlowFilter)。

这里是 projectileManager.createProjectilePool 函数：



```
public function createProjectilePool(maxProjectiles:int):void {  
    for (var projctileCtr:int = 0; projctileCtr < maxProjectiles; projctileCtr++) {  
        var tempProjectile:BasicBiltArrayProjectile=new BasicBiltArrayProjectile(0,799,0,799);  
        tempProjectile.animationList=projectileAnimationFrames;  
        tempProjectile.bitmapData = tempProjectile.animationList[0];  
        projectilePool.push(tempProjectile);  
    }  
}
```

projectilePool 和之前描述的 particle pool 非常相似。有两个数组来保存炮弹实例。projectilePool 持有不活动的炮弹，projectiles 数组持有使用中的活动的炮弹。

## 添加 newGame 和 newLevel 函数

BlasterMinse.as 的 newGame 函数设置新游戏的默认值。newLevel 函数使用一个简单的算法来设置新的一关玩家需要面对的敌人的颜色，速度和数量。

```
override public function newGame():void {  
    //cannot do this until the game has been added to the stage.  
    if (frameRateMultiplier==2) {  
        stage.quality = StageQuality.BEST;  
    }else {  
        stage.quality = StageQuality.MEDIUM;  
    }  
    trace("new game");  
    switchSystemState(STATE_SYSTEM_GAME_PLAY);  
    score = 0;  
    level = 0;  
    shield = 10;  
    gameOver = false;  
    playerExplosionParticles = [];  
    playerStarted = false;  
    customScoreBoardEventShield.value = shield.toString()  
    customScoreBoardEventScore.value = score.toString()  
    customScoreBoardEventLevel.value = level.toString()  
    dispatchEvent(customScoreBoardEventScore);  
    dispatchEvent(customScoreBoardEventShield);  
    dispatchEvent(customScoreBoardEventLevel);  
}
```

这个函数非常简单，但是我们要指出一些东西。首先，我们使用 frameRateModifier 值来设置 stage.quality。这是一个简单的例子，来使用分析过的信息来为每个玩家的机器改变游戏的品质。stagt.Quality 有 4 个设定：LOW,MEDIUM,HIGH 和 BEST。我们已经选择使用 MEDIUM，除非



setRendering 函数发现 frameRateModifier 因为玩家的机器被设置成 2。注意到类级别的重用的计分板更新事件首次使用在这个函数里。

这里是 newLevel 函数：

```
override public function newLevel():void {
    stage.focus = this;
    trace("new level");
    mineManager.mines = [];
    projectileManager.projectiles = [];
    stage.focus = this;
    level++;
    if (level > maxLevel) {
        level = 1;
    }
    levelColor = levelColors[level];
    spriteGlowFilter = new GlowFilter(levelColor, .75, 2, 2, 1.5, 2, false, false);
    createLevelbackground();
    mineManager.createLevelMines(spriteGlowFilter, level, levelColor);
    particleManager.createLevelParticles(spriteGlowFilter, levelColor);
    particleManager.createParticlePool(particleManager.particlePoolMax*_
    frameRateMultiplier);
    canvasBitmapGlowFilter.color = levelColor;
    canvasBitmap.filters = [canvasBitmapGlowFilter];
    player.x = 400;
    player.y = 400;
    player.nextX = player.x;
    player.nextY = player.y;
    playerStarted = true;
    player.bitmapData = player.animationList[0];
    render();
    dispatchEvent(new CustomEventLevelScreenUpdate(CustomEventLevelScreenUpdate._
    UPDATE_TEXT, String(level)));
    dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _
    Main.SOUND_MUSIC_IN_GAME, true, 999, 8, 1));
    customScoreBoardEventLevel.value = level.toString();
    dispatchEvent(customScoreBoardEventLevel);
}
```

newLevel 函数执行了很多的游戏逻辑。让我们一步一步的来看看这个函数的主要行为吧：

1.首先,将当前级别的值加 1.如果当前级别比 maxLevel 大, 就会设置回 1.



2.使用当前级别整数值，从 levelColors 数组中拿出一个颜色值，设置到类变量 levelColor。在 Mine 发光，Particle 发光，和这个级别的发光效果中会用到。这里是变量定义部分的 levelColors 数组。

```
private var levelColors:Array = [NaN, 0xff0000, 0x00ff00, 0x0000ff, 0xffff00, 0x00ffff, _  
0xffaa00, 0xaa00ff, 0x00ffaa, 0x00aaff];
```

3. 通过各自的创建函数将新的 levelColor 值来设置 spriteGlowFilter，mine，粒子和绘制背景资源。同时也创建的粒子池。

4. 整个 canvasBitmap 的特别的发光滤镜设置为 levelColor 然后将他应用到 cavasBitmap 的滤镜数组。

5. 玩家对象初始化在屏幕的中心同时被设置为 0 帧(朝上)

6. 调用渲染函数来保证等级转换动画消失的时候绘制屏幕。

7. 发出事件来设置 levelScreen 文字 播放游戏内的音乐 变更计分板上等级的文字。注意 scoreBoard 更新时使用类级别的共享事件。

注意我们只在 scoreBoard 更新的时候创建类级别的共享事件，但是如果我们想继续利用这些优化，在 levelScreen 更新中和 soundManager 这样做。

让我们看一看 createLevelBackground 函数：

```
private function createLevelbackground():void {  
    /*** background  
    drawingCanvas.filters = [spriteGlowFilter];  
    //draw symbol on background  
    drawingCanvas.graphics.clear();  
    drawingCanvas.graphics.lineStyle(2, 0xaa0000);  
    drawingCanvas.graphics.moveTo(6, 6);  
    drawingCanvas.graphics.lineTo(25, 6);  
    drawingCanvas.graphics.lineTo(25, 22);  
    drawingCanvas.graphics.lineTo(6, 22);  
    drawingCanvas.graphics.lineTo(6, 6);  
    drawingCanvas.graphics.moveTo(18, 8);  
    drawingCanvas.graphics.lineTo(18, 16);  
    drawingCanvas.graphics.lineTo(12, 16);  
    drawingCanvas.graphics.lineTo(12, 12);  
    drawingCanvas.graphics.moveTo(9, 24);  
    drawingCanvas.graphics.lineTo(9, 25);  
    drawingCanvas.graphics.moveTo(15, 24);  
    drawingCanvas.graphics.lineTo(15, 25);  
    drawingCanvas.graphics.moveTo(21, 24);
```



```
drawingCanvas.graphics.lineTo(21, 25);
//enlarge scale drawn symbol and place in center of the background
var scaleTranslateMatrix:Matrix = new Matrix();
scaleTranslateMatrix.scale(20, 20);
scaleTranslateMatrix.translate(100, 100);
var faceColorTransform:ColorTransform = new ColorTransform(1, 1, 1, .2 );
backgroundBitmapData.draw(drawingCanvas, scaleTranslateMatrix, faceColorTransform,
BlendMode.LAYER)
scaleTranslateMatrix = null;
faceColorTransform = null;
//draw box around background
drawingCanvas.graphics.clear();
drawingCanvas.graphics.lineStyle(2,0xffffffff);
drawingCanvas.graphics.drawRect(5, 5, 790, 790);
backgroundBitmapData.draw(drawingCanvas);
//draw 800 random stars on the background
for (var ctr:int = 0; ctr < 800; ctr ++ ) {
backgroundBitmapData.setPixel32(int(Math.random() * 799), _
int(Math.random() * 799), 0x0066ff);
}
}
```

backgroundBitmapData 实例用来让 blit 操作变得容易，而不是一个单纯的黑色背景，这样每帧，整个 800\*800 背景将会被拷贝到 canvasBitmapData

并且每个对象都会拷贝进去。这就是我们如何构造背景的概要。渲染的部分将会在 render 函数中调用。

- 1.首先，我们在 drawingCanvas 上绘制 mine 标记(在下一节中描述)。
2. 然后，我们用 scale 和转换矩阵放大并且移动整个标记到 backgroundBitmapData 的中心
3. 我们在整个 backgroundBitmapData 周围绘制一个矩形盒子，来给他一个薄边界。
4. 我们绘制 800 个随机的蓝色星星（单个像素）到 backgroundBitmapData.

现在，我们看一看 MineManager 类函数，createLevelMines.这个函数根据级别值创建一些 Mine 类实例。

```
public function createLevelMines(spriteGlowFilter:GlowFilter,level:int, _
levelColor:uint):void{
/** Mines look
mineBitmapData= new BitmapData(32, 32, true, 0x00000000);
var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();
drawingCanvas.graphics.clear();
```



```
drawingCanvas.graphics.lineStyle(2, 0xffffffff);
drawingCanvas.graphics.moveTo(6, 6);
drawingCanvas.graphics.lineTo(25, 6);
drawingCanvas.graphics.lineTo(25, 22);
drawingCanvas.graphics.lineTo(6, 22);
drawingCanvas.graphics.lineTo(6, 6);
drawingCanvas.graphics.moveTo(18, 8);
drawingCanvas.graphics.lineTo(18, 16);
drawingCanvas.graphics.lineTo(12, 16);
drawingCanvas.graphics.lineTo(12, 12);
drawingCanvas.graphics.moveTo(9, 23);
drawingCanvas.graphics.lineTo(9, 25);
drawingCanvas.graphics.moveTo(15, 23);
drawingCanvas.graphics.lineTo(15, 25);
drawingCanvas.graphics.moveTo(21, 23);
drawingCanvas.graphics.lineTo(21, 25);
spriteGlowFilter.color = levelColor;
mineBitmapData.draw(drawingCanvas);
mineBitmapData.applyFilter(mineBitmapData, mineBitmapData.rect, point0,
spriteGlowFilter);
tempBlitArrayAsset = new BlitArrayAsset();
mineAnimationFrames=tempBlitArrayAsset.createRotationBlitArrayFromBD_
(mineBitmapData, 1,90);
/** end look
//create mines for the level
mines = [];
for (var ctr:int=0;ctr<30+20*level;ctr++) {
var tempMine:Mine=new Mine(13,787,13,787);
//find a random roation value for the mine.
tempMine.dx=Math.cos(6.28*((Math.random()*360)-90)/360.0);
tempMine.dy = Math.sin(6.28 * ((Math.random() * 360) - 90) / 360.0);
if (level % 2 == 0) {
tempMine.y = 700
}else {
tempMine.y = 100
}
tempMine.x = 100;
tempMine.nextX = tempMine.x;
tempMine.nextY = tempMine.y;
tempMine.frame = int((Math.random() * 359)) ;
```



```
tempMine.animationList = mineAnimationFrames;
tempMine.bitmapData = tempMine.animationList[tempMine.frame];
tempMine.speed = (Math.random()*1)+2+level;
mines.push(tempMine);
}
}
```

根据等级值，为当前级别创建一些 Mine 实例。当前级别的 Mine 实例的数目，从 30 开始，每级别增加 20\*级别。这样高级别就会编一个非常难的游戏。

在每个级别创建之前，mine 将会在一个循环里被创建。mine 的数量从 30 开始，加上 20 乘以级别数：

```
for (var ctr:int=0;ctr<30+20*level;ctr++) {
... // interior loop code to create all mines for the level
}
```

接下来计算 mine 的速度，允许他每级增长：

```
tempMine.speed = (Math.random()*1)+2+level;
```

这个函数也创建了我们敌人 mine 的外观，一个 32\*32 像素的记号。这个记号跟在 Fireworks 里创建的玩家对象方法类似。同样绘制在 drawCanvas 的 Shape 实例上，放置到一个 BitmapData 实例 (mineBitmapData),并且用 BlitArrayAsset 类的实例来旋转。BlitArrayAsset.createRotationArray 调用中的 360 度旋转的 BitmapData 实例放置在一个全局的 mineAnimation 数组。每个新的 Mine 实例也被赋予了随机的 dx 和 dy 值。如果级别是一个奇数，我们从左上角(y=100)开始所有的 mine。如果级别是一个偶数，所有的 mine 从左下角(y=700)开始。

同样的，每个 mine 被赋予一个随机的 animationList 数组中的某一帧开始。每个 mine 的速度同样是一个基于等级值最大值的随机数。所有的这些 mine 计算对游戏玩家都是不是完全不可预知的。

接下来，我们会研究 particleManager.createParticlePool 函数。这里是函数完整的代码；你已经在前面的章节“优化对象池”里看过这段代码了：

```
public function createParticlePool(maxParticles:int):void {
particlePool = [];
particles = [];
for (var particleCtr:int=0;particleCtr<maxParticles;particleCtr++) {
var tempParticle:BasicBiltArrayParticle = new BasicBiltArrayParticle(0,799,0,799);
particlePool.push(tempParticle);
}
}
```

每个新级别都会有一个新的 Particle 实例的对象池创建。池中对象的数量是基于传入的 maxParticle 值。

现在，看一下 particleManager.createLevelParticle 函数：



```
public function createLevelParticles(spriteGlowFilter:GlowFilter, levelColor:uint):void {  
    var tempBlitArrayAsset:BlitArrayAsset = new BlitArrayAsset();  
    particleBitmapData = new BitmapData(8, 8, true, 0x00000000);  
    drawingCanvas.graphics.clear();  
    drawingCanvas.graphics.lineStyle(1, 0xffffffff);  
    drawingCanvas.graphics.drawRect(3, 3, 2, 2);  
    particleBitmapData.draw(drawingCanvas);  
    spriteGlowFilter.color = levelColor;  
    particleBitmapData.applyFilter(particleBitmapData, particleBitmapData.rect, _  
    point0, spriteGlowFilter);  
    tempBlitArrayAsset= new BlitArrayAsset();  
    tempBlitArrayAsset.createFadeOutBlitArrayFromBD(particleBitmapData, 30);  
    particleAnimationFrames = tempBlitArrayAsset.tileList;  
}
```

粒子的外观是基于 Mine 的 levelColor。颜色是在 BlasterMine 的 spriteGlowFilter 实例里传入这个函数的。粒子用 BlitArrayAsset 类的 createFadeOutArrayFrameBD 来创建一个消失帧的数组。在连续运行的帧数里，粒子从爆炸的重心移动开时会变淡然后消失。

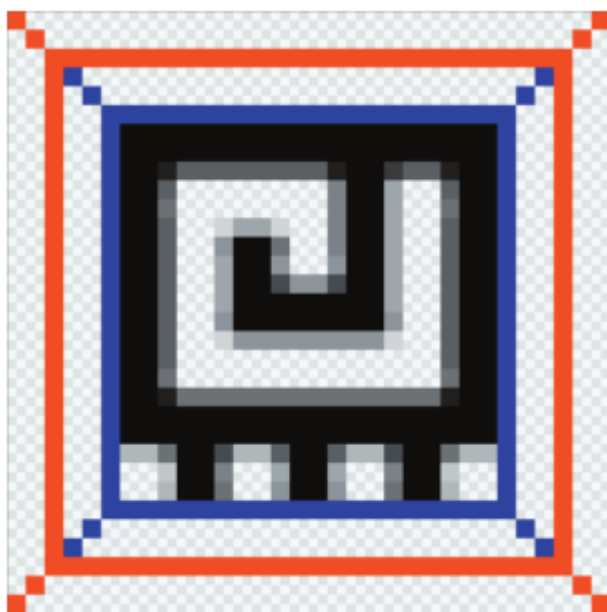


Figure 11-5. The 32 \_ 32 Mine graphic blown up in design mode

## 更新游戏循环和内置状态机



虽然 Blaster Mine 游戏循环和内置状态机和前面章节的一样，这里我们把他更新成可以使用基于时间的步进计时器。需要一个特别为你的游戏复写 Game.as 类的 runGameTimeBase 函数。添加这个函数到 BlasterMines.as 文件:

runGameTimeBased 函数添加到 Game.as 基类，用于替换游戏中的 runGame 函数，这样就可以使用基于时间的计时器。

```
override public function
runGameTimeBased(paused:Boolean=false,timeDifference:Number=1):void {
    if (!paused) {
        systemFunction(timeDifference);
    }
}
```

暂停功能和基于时间的计时器的核心部分在这个函数里。Paused 布尔值和 timeDifference 值同时从 Main 中传入这个函数。如果有必要当前 systemFunction 运行并且使用 timeDifference。

switchSystemState 函数用来在两个基本的游戏状态中切换：GAMEPLAY 和 PLAYEREXPLODE。

```
private function switchSystemState(stateval:int):void {
    lastSystemState = currentSystemState;
    currentSystemState = stateval;
    switch(stateval) {
        case STATE_SYSTEM_GAME_PLAY:
            systemFunction = systemGamePlay;
            break;
        case STATE_SYSTEM_PLAYER_EXPLODE :
            systemFunction = systemPlayerExplode;
            break;
    }
}
```

systemGamePlay 函数通过将 timeDifference 变量传入 update 函数来让它生效。

```
private function systemGamePlay(timeDifference:Number=0):void {
    trace("in game play");
    update(timeDifference);
    checkCollisions();
    render();
    updateScoreBoard();
    checkforEndLevel();
    checkforEndGame();
}
```



这个游戏的 checkForEndGame 函数是独特的。当玩家的护盾值消耗完，游戏就结束了。当 shield 变量小于 0，checkCollision 函数就会设置 gameOver 布尔值为 true。玩家有一个特别的爆炸函数，显示为 300 个粒子。粒子的外观重用了炮弹的 BitmapData。当玩家执行爆炸时，内置的游戏状态机设置为运行 STATE\_SYSTEM\_PLAYEREXPLODE 状态而不是普通的 STATE\_SYSTEM\_GAMEPLAY 状态。

```
private function checkforEndGame():void {
    if (gameOver ) {
        dispatchEvent(new CustomEventSound(CustomEventSound.STOP_SOUND, _
        Main.SOUND_MUSIC_IN_GAME, true));
        dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _
        Main.SOUND_PLAYER_EXPLODE, false, 1, 8, 1));
        canvasBitmapData.copyPixels(backgroundBitmapData, new Rectangle(0,0,32,32), _
        player.point);
        createPlayerExplode(player.x + 16, player.y + 16, 300);
        playerStarted = false;
        switchSystemState(STATE_SYSTEM_PLAYER_EXPLODE);
    }
}
```

checkForEndLevel 函数决定玩家是否已经进入下一个游戏级别。当所有的 mines 已经被摧毁当前级别结束。我们不希望级别结束得太突然，所以我们等到知道 24 个例子残留在屏幕上时，开始新的级别。玩家背景被拷贝到当前位置上，玩家就被从屏幕上抹去。

```
private function checkforEndLevel():void {
    if (mineManager.mines.length == 0 && particleManager.particles.length < 25 ) {
        dispatchEvent(new CustomEventSound(CustomEventSound.STOP_SOUND, _
        Main.SOUND_MUSIC_IN_GAME, true));
        disposeAll();
        playerStarted = false;
        //erase player from screen
        canvasBitmapData.copyPixels(backgroundBitmapData, new Rectangle(0,0,32,32), _
        player.point);
        dispatchEvent(new Event(NEW_LEVEL));
    }
}
```

当 gameOver 变量被设置为 true，checkForEndGame 函数将会从 currentSystemState 切换到 STATE\_SYSTEM\_PLAYEREXPLODE。在 STATE\_SYSTEM\_PLAYEREXPLODE 状态里，systemPlayerExplode 函数设置为 systemFunction。这个函数会每帧持续的调用更新和渲染函数，直到玩家爆炸产生的所有粒子都从屏幕上消失。

```
private function systemPlayerExplode(timeDifference:Number=0):void {
    update(timeDifference);
}
```



```
render();
if (playerExplosionParticles.length == 0) {
    playerExplodeComplete();
}
}
```

当没有更多的玩家爆炸粒子剩余了，systemPlayerExplode 状态函数调用 playerExplodeComplete 函数。当 Main 收到这个事件时，会更改框架的系统状态。同时，调用 disposeAll 函数。它会清理所有的对象实例并且准备好一个新游戏。

```
private function playerExplodeComplete():void {
    dispatchEvent(new Event(GAME_OVER));
    disposeAll();
}
```

## 添加更新，自动射击，渲染和撞击函数

更新，自动射击，渲染和撞击函数跟你在前面的游戏里看到的十分相似。在本章前面我们也描述大部分的功能。我们会过一遍高亮和任何前面没有提到的新的主题。

```
private function update(timeDifference:Number = 0):void {
    //time based movement modifier calculation
    var step:Number = (timeDifference / 1000)*timeBasedUpdateModifier;
    /*** auto fire
    autoShoot(step);
    if (playerStarted) {
        player.update(this.mouseX + canvasBitmap.scrollRect.x, this.mouseY + ~cc
        canvasBitmap.scrollRect.y, 20,step);
    }
    for each (tempMine in mineManager.mines) {
        tempMine.update(step);
        tempMine.updateFrame(5);
    }
    particleManager.particleCount = particleManager.particles.length-1
    for (var ctr:int = particleManager.particleCount; ctr >= 0; ctr--) {
        tempParticle = particleManager.particles[ctr];
        if (tempParticle.update(step)) { //return true if particle is to be removed
            tempParticle.frame = 0;
            particleManager.particlePool.push(tempParticle);
            particleManager.particles.splice(ctr,1);
        }
    }
    var projectileLength:int = projectileManager.projectiles.length - 1;
```



```
for (ctr=projectileLength;ctr>=0;ctr--) {  
    tempProjectile = projectileManager.projectiles[ctr];  
    if (tempProjectile.update(player.xMove, player.yMove,step)) {  
        // returns true if needs to be removed  
        tempProjectile.frame = 0;  
        projectileManager.projectilePool.push(tempProjectile);  
        projectileManager.projectiles.splice(ctr,1);  
    }else {  
        tempProjectile.updateFrame(1);  
    }  
}  
  
for (ctr = playerExplosionParticles.length-1; ctr >= 0; ctr--) {  
    tempParticle = playerExplosionParticles[ctr];  
    if (tempParticle.update()) { //return true if particle is to be removed  
        tempParticle=null  
        playerExplosionParticles.splice(ctr,1);  
    }  
}  
}
```

更新函数循环并且更新我们所有的游戏对象如下：

1. 更新做的第一件事是计算当前帧率的修正器的步进值。我们已经在本章的前面部分详细的讨论过了，但是澄清一下，步进值是当前帧率下实际计算出来的我们希望每个游戏对象移动距离的百分比，这个值也是根据基于时间步进计时器的结果。

2. 玩家飞船自动发射炮弹。炮弹是从 projectileManager.projectilePool 中产生，从每三帧里飞船的位置发出。发射炮弹调用的是 autoShoot 函数。

3. 传入当前的 mouseX 和 mouseY 值，一个玩家相对于鼠标的延迟值来更新玩家对象。延迟值越大，飞船跟随鼠标的速度越慢。

4. 更新每个 mine 的位置。然后，传入这 5 到 mine 的 updateFrame 函数来更新每个 mine 的帧数，这样就会使 mine 每帧忽略 5 帧动画从而旋转得更快。这个 5 是随意的。它只是告诉 Mine 每次旋转 5 度。这个值可以设置为任意一个你希望创建的不同 Mine 旋转速率。

5.更新炮弹

6.如果玩家爆炸了，更新玩家的爆炸粒子。

注意每个更新函数需要传入步进值来用计算当前帧移动对象的距离。

下一步运行，每帧都被调用的 autoShoot 函数。如果 projectileManager.projectilePool 中有剩余，在每三帧延迟后自动发射炮弹。



```
private function autoShoot(step:Number):void {
    projectileManager.projectilePoolCount = projectileManager.projectilePool.length - 1;
    mineManager.mineCount = mineManager.mines.length;
    if (projectileManager.lastProjectileShot > 3 && projectileManager.projectilePoolCount_
    > 0 && playerStarted && mineManager.mineCount > 0) {
        dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _
        Main.SOUND_PLAYER_SHOOT, false, 0, 8, 1));
        tempProjectile = projectileManager.projectilePool.pop();
        var projectileRadians:Number = (player.frame / 360) * 6.28;
        //+ 16 to get it to the center
        tempProjectile.x=(player.point.x+16)+Math.cos(projectileRadians);
        tempProjectile.y =(player.point.y+16) + Math.sin(projectileRadians);
        tempProjectile.x = player.x+16;
        tempProjectile.y = player.y + 16;
        tempProjectile.nextX = tempProjectile.x;
        tempProjectile.nextY = tempProjectile.y;
        tempProjectile.dx = rotationVectorList[player.frame].x;
        tempProjectile.dy = rotationVectorList[player.frame].y;
        tempProjectile.speed = 5;
        tempProjectile.frame = 0;
        tempProjectile.bitmapData = tempProjectile.animationList[0];
        projectileManager.projectiles.push(tempProjectile);
        projectileManager.lastProjectileShot=0;
    }else {
        projectileManager.lastProjectileShot+=step;
    }
}
```

同时注意我们添加步进值到 `projectileManager.lastProjectileShot` 计时器而不是标准的每帧传的数字 1。这样，我们就保证玩家发射的速率是按照步进计时器而不是帧率来计时的。如果我们直接将它直接绑定到帧率上，速度较快的机器上的用户就会被机器较慢的用户开火得更频繁。在我们的游戏里，无论游戏运行在什么样的帧率上，每个玩家每秒都会发射同样数量的炮弹。

现在，我们看一下 `checkCollision` 函数：

```
private function checkCollisions():void {
    mineManager.mineCount = mineManager.mines.length - 1;
    projectileManager.projectileCount = projectileManager.projectiles.length - 1;
    mines: for (var mineCtr:int = mineManager.mineCount; mineCtr >= 0; mineCtr--) {
        tempMine=mineManager.mines[mineCtr];
        tempMine.point.x=tempMine.x;
        tempMine.point.y=tempMine.y;
```



```
projectiles: for (var projectileCtr:int=projectileManager.projectileCount;_
projectileCtr>=0;projectileCtr--) {
tempProjectile=projectileManager.projectiles[projectileCtr];
tempProjectile.point.x=tempProjectile.x;
tempProjectile.point.y=tempProjectile.y;
//use pixel hit because circle circle for
//these was causing false negatives on ship shield hit
if (tempProjectile.bitmapData.hitTest(tempProjectile.point,255,_
tempMine.bitmapData,tempMine.point,255)) {
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _
Main.SOUND_MINE_EXPLODE, false, 0, 8, 1));
createExplode(tempMine.x+16, tempMine.y+16, particleManager._
particlesPerExplode*frameRateMultiplier);
tempMine=null;
mineManager.mines.splice(mineCtr, 1);
tempProjectile.frame = 0;
projectileManager.projectilePool.push(tempProjectile);
projectileManager.projectiles.splice(projectileCtr,1);
score += 5 * level;
break mines;
break projectiles;
}
}
if (circleCheck(player.point.x, tempMine.point.x, player.point.y, _
tempMine.point.y, 12, 12)) {
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _
Main.SOUND_MINE_EXPLODE, false, 0, 8, 1));
createExplode(tempMine.x+16, tempMine.y+16, particleManager._
particlesPerExplode*frameRateMultiplier);
tempMine=null;
mineManager.mines.splice(mineCtr, 1);
score += 5 * level;
//trace("hit");
if (!player.shieldRender) {
//trace("start shield");
shield--;
if (shield < 0) {
shield = 0;
gameOver = true;
}else {
```



```
dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _
Main.SOUND_PLAYER_HIT, false, 0, 8, 1));
player.shieldRender = true;
player.shieldCount = 0;
}
}else {
//trace("shield already started");
}
}
}
}
```

现在让我们一步一步的看看前面的代码里干了些什么：

1. 首先，我们循环所有的 mine 并且对他们进行针对所有的炮弹进行圆到圆的数学检查。如果一个碰撞发生，就移除他们，并且更新玩家分数。粒子和炮弹被放回各自的缓冲池以备重用。
2. 玩家飞船和 mine 用 BitmapData 进行完整像素的碰撞检测。我们使用这种类型的碰撞检测是因为玩家飞船是一个三角形，使用圆对圆的检测太不准确了。
3. 如果玩家飞船被一个 Mine 实例碰到，它的护盾就会显示几帧。如果 player.shieldCount 比 1 小，gameOver 布尔值就会设置为 true。

```
private function render():void {
canvasBitmapData.lock();
canvasBitmapData.copyPixels(backgroundBitmapData, backgroundBitmapData.rect, point0);
if (playerStarted) {
player.render(canvasBitmapData);
}
for each (tempMine in mineManager.mines) {
tempMine.render(canvasBitmapData);
}
for each (tempParticle in particleManager.particles) {
tempParticle.render(canvasBitmapData);
}
for each (tempProjectile in projectileManager.projectiles) {
tempProjectile.render(canvasBitmapData);
}
for each (tempParticle in playerExplosionParticles) {
tempParticle.render(canvasBitmapData);
}
if (player.shieldRender) {
//trace("render shield");
}
```



```
canvasBitmapData.copyPixels(player.shieldBitmapData, player._
shieldBitmapData.rect, player.point);
player.shieldCount++;
if (player.shieldCount > player.shieldLife) {
player.shieldCount = 0;
player.shieldRender = false;
}
}
canvasBitmapData.unlock();
if (playerStarted) {
canvasRect.x = player.x - 200;
canvasRect.y = player.y - 200;
if (canvasRect.x < 0) canvasRect.x = 0;
if (canvasRect.y < 0) canvasRect.y = 0;
if (canvasRect.x > 399) canvasRect.x = 399;
if (canvasRect.y > 399) canvasRect.y = 399;
canvasBitmap.scrollRect = canvasRect;
}
}
```

渲染函数简单的锁定我们的容器，循环所有的游戏对象并且 blit 他们到 canvasBitmapData。最后一个任务是确保 canvasBitmapData 的 scrollRect 被定位到正确的位置来模拟滚屏。

## 添加辅助函数

许多剩余的辅助函数跟你前面在其他游戏里看到的十分相似。跟前面一样，我们描述主要的差别和新函数。

```
private function updateScoreBoard():void {
customScoreBoardEventScore.value = score.toString();
customScoreBoardEventShield.value = shield.toString()
customScoreBoardEventParticlePool.value = String(particleManager.particlePool.length);
customScoreBoardEventParticleActive.value = String(particleManager.particles.length);
customScoreBoardEventProjectilePool.value = String(projectileManager._
projectilePool.length);
customScoreBoardEventProjectileActive.value = String(projectileManager._
projectiles.length);
dispatchEvent(customScoreBoardEventScore);
dispatchEvent(customScoreBoardEventShield);
dispatchEvent(customScoreBoardEventParticlePool);
dispatchEvent(customScoreBoardEventParticleActive);
dispatchEvent(customScoreBoardEventProjectilePool);
}
```



```
dispatchEvent(customScoreBoardEventProjectileActive);
}
```

The ScoreBoard is updated using our new set of global custom Events.

```
private function addToScore(val:Number):void {
    score += val;
}
```

The addToScore function simply accepts in a value to add to the current score.

```
private function circleCheck(x1:Number, x2:Number, y1:Number, y2:Number, _
    radius1:Number, radius2:Number):Boolean {
    var dx:Number = x2 - x1;
    var dy:Number = y2 - y1;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    return dist < radius1 + radius2
}
```

circleCheck 函数传入两个对象的位置和他们的半径，如果他们相交，意味着碰撞，返回一个 true。

```
private function createExplode(xval:Number,yval:Number,parts:int):void {
    for (var explodeCtr:int=0;explodeCtr<parts;explodeCtr++) {
        particleManager.particlePoolCount = particleManager.particlePool.length-1;
        if (particleManager.particlePoolCount > 0) {
            tempParticle=particleManager.particlePool.pop();
            tempParticle.lifeDelayCount=0;
            tempParticle.x=xval;
            tempParticle.y = yval;
            tempParticle.nextX=xval;
            tempParticle.nextY=yval;
            tempParticle.speed = (Math.random() * 3) + 1;
            tempParticle.frame = 0;
            tempParticle.animationList = particleManager.particleAnimationFrames;
            tempParticle.bitmapData = tempParticle.animationList[tempParticle.frame];
            var randInt:int = int(Math.random() * 359);
            tempParticle.dx = rotationVectorList[randInt].x;
            tempParticle.dy = rotationVectorList[randInt].y;
            tempParticle.lifeDelay = int(Math.random() * 10);
            particleManager.particles.push(tempParticle);
        }
    }
}
```

createPlayerExplode 函数用从 particlePool 中拿出的 Particel 创建一个爆炸(如果有足够的剩余在对象池中)



```
private function createPlayerExplode(xval:Number,yval:Number,parts:int):void {
    for (var explodeCtr:int=0;explodeCtr<parts;explodeCtr++) {
        var tempParticle:BasicBiltArrayParticle = new BasicBiltArrayParticle(0,799,0,799);
        tempParticle.lifeDelayCount=0;
        tempParticle.x=xval;
        tempParticle.y = yval;
        tempParticle.nextX=xval;
        tempParticle.nextY=yval;
        tempParticle.speed = (Math.random() * 10) + 1;
        tempParticle.frame = 0;
        tempParticle.animationList = projectileManager.projectileAnimationFrames;
        tempParticle.bitmapData = tempParticle.animationList[tempParticle.frame];
        var randInt:int = int(Math.random() * 359);
        tempParticle.dx = rotationVectorList[randInt].x;
        tempParticle.dy = rotationVectorList[randInt].y;
        tempParticle.lifeDelay = int(Math.random() * 5);
        playerExplosionParticles.push(tempParticle);
    }
}
```

createPlayerExplode 函数创建一个玩家飞船用的爆炸。我们使用 projectileAminiation 的第一帧，这是个 BitmapData 实例组成的数组。我们使用一个 Projectile BitmapData 而不是创建一个新的粒子因为它已经是跟玩家一样的颜色了。

```
private function disposeAll():void {
    particleManager.particleCount = particleManager.particles.length-1
    // remove particle pool
    for (var ctr:int = particleManager.particleCount; ctr >= 0; ctr--) {
        tempParticle = particleManager.particles.pop();
        tempParticle.frame = 0;
        particleManager.particlePool.push(tempParticle);
    }
    //remove particle animation
    for (ctr= 0; ctr < particleManager.particleAnimationFrames.length; ctr++) {
        particleManager.particleAnimationFrames[ctr].dispose();
    }
    particleManager.particleAnimationFrames = null;
    particleManager.particleBitmapData = null;
    particleManager.particlePool = null;
    trace("disposed");
    //remove mines
}
```



```
for (ctr= 0; ctr < mineManager.mineAnimationFrames.length; ctr++) {  
    mineManager.mineAnimationFrames[ctr].dispose();  
}  
mineManager.mineAnimationFrames = null;  
mineManager.mines = null;  
mineManager.mineBitmapData = null;  
if (gameOver) {  
    playerExplosionParticles = null;  
}  
}
```

disposeAll 函数运行在每关之间并且在每个游戏结束后，通过释放他们进垃圾处理流程来节省内存。

## 完整的 GameFrameWork 类

为了参考在这节里，我们将整段的新的 com.efg.framework.GameFrameWork.as 文件包含进来。在这章里，这个文件已经修改很多。我们建议你用它来替代现有的 GameFrameWork.as 文件。在前面的小节里我们没有讲到每个变化的的所有代码。注意第 11 章里注释显示出来的变化。

```
package com.efg.framework  
{  
    import flash.display.Bitmap;  
    import flash.display.BitmapData;  
    import flash.display.MovieClip;  
    import flash.events.Event;  
    import flash.events.KeyboardEvent;  
    import flash.geom.Point;  
    import flash.text.TextFormat;  
    import flash.utils.getTimer;  
    import flash.utils.Timer;  
    import flash.events.TimerEvent;  
    public class GameFrameWork extends MovieClip {  
        public static const EVENT_WAIT_COMPLETE:String = "wait complete";  
        //added in chapter 11  
        public static const KEY_MUTE:int = 77; // added chapter 11  
        public static const KEY_PAUSE:int = 80; //added chapter 11  
        public var paused:Boolean = false;  
        public var pausedScreen:BasicScreen;  
        public var systemFunction:Function;  
        public var currentSystemState:int;  
        public var nextSystemState:int;  
        public var lastSystemState:int;
```



```
public var appBackBitmapData:BitmapData;
public var appBackBitmap:Bitmap; ;
public var frameRate:int;
public var timerPeriod:Number;
public var gameTimer:Timer;
public var titleScreen:BasicScreen;
public var gameOverScreen:BasicScreen;
public var instructionsScreen:BasicScreen;
public var levelInScreen:BasicScreen;
public var scoreBoard:ScoreBoard;
public var scoreBoardTextFormat:TextFormat;
public var screenTextFormat:TextFormat;
public var screenButtonFormat:TextFormat;
public var levelInText:String;
public var soundManager:SoundManager;
//chapter 11 added
public var frameCounter:FrameCounter = new FrameCounter();
public var lastTime:Number;
public var timeDifference:Number
public var game:Game;
//waitTime is used in conjunction with the STATE_SYSTEM_WAIT state
// it suspends the game and allows animation or other processing to finish
public var waitTime:int;
public var waitCount:int = 0;
//added chapter 11
public var frameRateProfiler:FrameRateProfiler;
public function GameFrameWork() {
soundManager = new SoundManager(); // moved;
}
//function added in chapter 11
public function addedToStage(e:Event = null):void {
stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownListener);
this.focusRect=false; // added chapter 11
stage.focus = stage; // added chapter 11
trace("gamFrameWork added to stage");
}
public function init():void {
trace("inner init stub");
}
public function frameRateProfileComplete(e:Event):void {
```



```
// stub
}

public function setApplicationBackGround(width:Number, height:Number, _
isTransparent:Boolean = false, color:uint = 0x000000):void {
    appBackBitmapData = new BitmapData(width, height, isTransparent, color);
    appBackBitmap = new Bitmap(appBackBitmapData);
    addChild(appBackBitmap);
}

//changed for chapter 11
public function startTimer(timeBasedAnimation:Boolean=false):void {
    stage.frameRate = frameRate;
    if (timeBasedAnimation) {
        lastTime = getTimer();
        addEventListener(Event.ENTER_FRAME, runGameEnterFrame);
    }else{
        timerPeriod = 1000 / frameRate;
        gameTimer=new Timer(timerPeriod);
        gameTimer.addEventListener(TimerEvent.TIMER, runGame);
        gameTimer.start();
    }
}

565
public function runGame(e:TimerEvent):void {
    systemFunction();
    frameCounter.countFrames();
}

public function runGameEnterFrame(e:Event):void {
    timeDifference = getTimer() - lastTime
    lastTime = getTimer();
    systemFunction();
    frameCounter.countFrames();
}

public function switchSystemState(stateval:int):void {
    lastSystemState = currentSystemState;
    currentSystemState = stateval;
    trace("currentSystemState=" + currentSystemState)
    switch(stateval) {
    case FrameWorkStates.STATE_SYSTEM_WAIT:
        systemFunction = systemWait;
        break;
```



```
case FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE:
systemFunction = systemWaitForClose;
break;
case FrameWorkStates.STATE_SYSTEM_TITLE:
systemFunction = systemTitle;
break;
case FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS:
systemFunction = systemInstructions;
break;
case FrameWorkStates.STATE_SYSTEM_NEW_GAME:
systemFunction = systemNewGame;
break;
case FrameWorkStates.STATE_SYSTEM_NEW_LEVEL:
systemFunction = systemNewLevel;
break;
case FrameWorkStates.STATE_SYSTEM_LEVEL_IN:
systemFunction = systemLevelIn;
break;
case FrameWorkStates.STATE_SYSTEM_GAME_PLAY:
systemFunction = systemGamePlay;
break
case FrameWorkStates.STATE_SYSTEM_GAME_OVER:
systemFunction = systemGameOver;
break;
}
}
public function systemTitle():void {
addChild(titleScreen);
titleScreen.addEventListener(CustomEventButtonId.BUTTON_ID, _
okButtonClickListener, false, 0, true);
switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);
nextSystemState = FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS;
}
public function systemInstructions():void {
//trace("system instructions");
addChild(instructionsScreen);
instructionsScreen.addEventListener(CustomEventButtonId.BUTTON_ID, _
okButtonClickListener, false, 0, true);
switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);
nextSystemState = FrameWorkStates.STATE_SYSTEM_NEW_GAME;
```



```
}
public function systemNewGame():void {
    addChild(game);
    game.addEventListener(CustomEventScoreBoardUpdate.UPDATE_TEXT, _
scoreBoardUpdateListener, false, 0, true);
    game.addEventListener(CustomEventLevelScreenUpdate.UPDATE_TEXT, _
levelScreenUpdateListener, false, 0, true);
    game.addEventListener(CustomEventSound.PLAY_SOUND, soundEventListener, _
false, 0, true);
    game.addEventListener(Game.GAME_OVER, gameOverListener, false, 0, true);
    game.addEventListener(Game.NEW_LEVEL, newLevelListener, false, 0, true);
    game.newGame();
    switchSystemState(FrameWorkStates.STATE_SYSTEM_NEW_LEVEL);
}
public function systemNewLevel():void {
    game.newLevel();
    switchSystemState(FrameWorkStates.STATE_SYSTEM_LEVEL_IN);
}
public function systemLevelIn():void {
    addChild(levelInScreen);
    waitTime = 30;
    switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT);
    nextSystemState = FrameWorkStates.STATE_SYSTEM_GAME_PLAY;
    addEventListener(EVENT_WAIT_COMPLETE, waitCompleteListener, false, 0, true);
}
public function systemGameOver():void {
    removeChild(game);
    addChild(gameOverScreen);
    gameOverScreen.addEventListener(CustomEventButtonId.BUTTON_ID, _
okButtonClickListener, false, 0, true);
    switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);
    nextSystemState = FrameWorkStates.STATE_SYSTEM_TITLE;
}
public function systemGamePlay():void {
    game.runGame();
}
public function systemWaitForClose():void {
    //do nothing
}
public function systemWait():void {
```



```
waitCount++;
if (waitCount > waitTime) {
waitCount = 0;
dispatchEvent(new Event(EVENT_WAIT_COMPLETE));
}
}

public function okButtonClickListener(e:CustomEventButtonId):void {
switch(e.id) {
case FrameWorkStates.STATE_SYSTEM_TITLE:
removeChild(titleScreen);
titleScreen.removeEventListener(CustomEventButtonId.BUTTON_ID, _
okButtonClickListener);
break;
case FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS:
removeChild(instructionsScreen);
instructionsScreen.removeEventListener(CustomEventButtonId.BUTTON_ID, _
okButtonClickListener);
break;
case FrameWorkStates.STATE_SYSTEM_GAME_OVER:
removeChild(gameOverScreen);
gameOverScreen.removeEventListener(CustomEventButtonId.BUTTON_ID, _
okButtonClickListener);
break;
}
trace("next system state=" + nextSystemState);
switchSystemState(nextSystemState);
}

public function scoreBoardUpdateListener(e:CustomEventScoreBoardUpdate):void {
scoreBoard.update(e.element, e.value);
}

public function levelScreenUpdateListener(e:CustomEventLevelScreenUpdate):void {
levelInScreen.setText(levelInText + e.text);
}

public function gameOverListener(e:Event):void {
switchSystemState(FrameWorkStates.STATE_SYSTEM_GAME_OVER);
game.removeEventListener(CustomEventScoreBoardUpdate.UPDATE_TEXT, _
scoreBoardUpdateListener);
game.removeEventListener(CustomEventLevelScreenUpdate.UPDATE_TEXT, _
levelScreenUpdateListener);
game.removeEventListener(CustomEventSound.PLAY_SOUND, soundEventListener);
```



```
game.removeEventListener(Game.GAME_OVER, gameOverListener);
game.removeEventListener(Game.NEW_LEVEL, newLevelListener);
}
public function newLevelListener(e:Event):void {
switchSystemState(FrameWorkStates.STATE_SYSTEM_NEW_LEVEL);
}
public function waitCompleteListener(e:Event):void {
switch(lastSystemState) {
case FrameWorkStates.STATE_SYSTEM_LEVEL_IN:
removeChild(levelInScreen);
break
}
removeEventListener(EVENT_WAIT_COMPLETE, waitCompleteListener);
switchSystemState(nextSystemState);
}
public function soundEventListener(e:CustomEventSound):void {
if (e.type == CustomEventSound.PLAY_SOUND) {
//trace("play sound");
soundManager.playSound(e.name, e.isSoundTrack, e.loops, e.offset, e.volume );
}else {
soundManager.stopSound(e.name, e.isSoundTrack);
}
}
public function keyDownListener(e:KeyboardEvent):void {
trace("key down: " + e.keyCode);
switch(e.keyCode) {
case KEY_PAUSE:
//pause key pressed
pausedKeyPressedHandler();
break;
case KEY_MUTE:
muteKeyPressedHandler();
break;
}
}
public function pausedScreenClickListener(e:Event):void {
removeChild(pausedScreen);
pausedScreen.removeEventListener(CustomEventButtonId.BUTTON_ID, _
okButtonClickListener);
trace("clicked");
paused = false;
```



```
stage.focus = game;
}
public function pausedKeyPressedHandler():void {
    trace("handle pause");
    addChild(pausedScreen);
    pausedScreen.addEventListener(CustomEventButtonId.BUTTON_ID, _
    pausedScreenClickListener, false, 0, true);
    paused = true;
}
public function muteKeyPressedHandler():void {
    soundManager.muteSound();
}
}
}
```

这个是完整的 GameFrameWork.as 文件。我们在第十二章的 Mochi 集成里讲到预加载时会添加很多到里面。

## 测试！

如果你已经做了这么多，你需要一个 10 级的后复古 blaster 游戏。测试内建在你发布系统里的选项。FrameRateProfiler 应该会马上启动，当它完成的时候，应该会有一个简单的标题屏幕，右侧有一个分数板。一旦你点击了屏幕上的按钮，游戏就会开始。使用鼠标移动，你的飞船就会跟着鼠标并且自动开火。

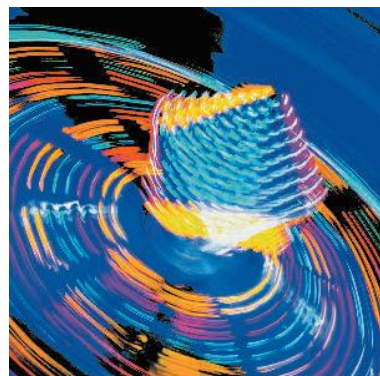
计分板下面的帧计数器里的帧率运行得如何？是否在你的电脑上游戏运行得缓慢？你可以回到设置在 Main.as 里的 FrameRateProfiler 并且调整分析器，用更多的对象进行测试是否游戏运行得太慢。调整分析将会帮助游戏设置 stage.quality 和 frameRateMultiplier 为一个低配置设定。

记住，你想要校准分析过的帧率到游戏在你机器上运行的帧率，以便他们相符。这样就保证分析器同样在其他玩家的机器上运行。

## 摘要

在这一章，我们已经以很快的速度讲了很多。我们讲到了 blitted 游戏的很多优化方法，这些方法能应用到很多其他的 Flash 游戏上。我们也创建了一个新的基于时间的步进计时器，为框架添加了暂停和静音功功能，并且完成了另一个完整的游戏。

在完成这次马拉松般的会议后，你可以放松一下，但是还不要退出。在第十二章，我们会添加一个 FlexSDK 和 FlashIDE 版本的 preloader 到 Main.as 类，来完善框架。我们也会讲到如何创建一个病毒式推广的 Flash 游戏。最后，我们会探索添加一个 Mochi 广告和一个 Mochi 高分到你的新完成的创意游戏里。



## 第十二章

# 制作一个 Viral Game：隧道惊魂

Viral Game 游戏是指某一种类型的游戏，比如沙龙曼蛇这种射击游戏

在前面 11 个章节,我们深入学习了九个不同游戏，并且对第一部分提到的游戏框架进行了多处更新。在这一章节中，我们还将继续学习制作一个网页版的病毒式游戏。接下来的这个游戏中我们还会应用的这个框架。我们还将介绍基于 AS3 的 Flex SDK 和 Flash IDE 的预加载方式。为了能完善这个框架，我们将介绍怎样在游戏中使用麻球(Mochi)媒体服务框架来展示一些广告和显示最高分。

我们将使用这两天来我们已经整理出的游戏构架来创建隧道惊魂游戏：这一架构可以为我们的读者描摹出了创作 Viral Game 的可行方案。Viral Game 的开发者有很多种盈利的手段，我们也将在这一章探讨其中的一些，比如授权盈利（使用 FlashGameLicense.com）以及自建主机运营等等，不过我们将重点深入最流行的几种手段的其中之一：使用麻球(Mochi)媒体服务。麻球网为广大 viral game 的开发者开创了一种盈利方式，我们的游戏就要用到其中的广告预加载工具和高分排行函数。

## 定义 viral web game

病毒网页游戏（viral web games）？听起来挺恐怖的。从某个角度讲，确实是这样的。病毒网页游戏在过去的几年曾经是非常风靡的传统游戏。事实上，一些人认为病毒网页游戏的兴起会成为导致传统游戏行业走向逐渐消亡的因素之一。



那么什么是 viral web games 呢？下面是 viral web games 游戏的几个显著的特征：

这类游戏可以借助插件直接在网络浏览器中玩且并不依赖于所在的网页。

这类游戏通常都是由 Flash Shockwave，Java 编写。近来甚至开始用 Silverlight,HTML5 和 Unity。

这类游戏可以很方便地嵌入网站或者通过网站接口导入。这也意味着整个游戏往往是一个单一的文件，或者以其他方便嵌入到网站的形式存在。

Viral games 可以放在在线游戏类网站上玩，也可以整合到其他游戏门户网站中。

最后，这类游戏可以通过嵌入广告，利用高分积分榜，或者其他诸如虚拟物品交易的方式盈利。

不过，仅仅知道什么是 viral web game 并不能教会你怎么做好它。本章接下来的部分会教给你一些法子，使你可以通过一个 viral web game 去创造和维持一点小收入。

## 发布一个网页游戏

在你创建好了游戏之后,接下来就是将它公布于众了。你可能很想马上发布出去，但这样并不好，我建议慢点，这里还有几点要点供你参考。

## 使用你自己的网站

推广你的 viral web game 的最快和最简单的方式莫过于你自己建站运营它了。恰好自主运营也是最灵活的运营方式。当你创建了一个网站，嵌入了你的 swf 游戏文件之后，你还得做点事情才能让你的游戏声名在外。

## 使用社交新闻类网站

使用社交新闻类网站，比如 Digg.com，是推广你的游戏的一个绝佳途径。在 Digg.com 上发布你的游戏非常简单，只要去 <http://digg.com> 创建一个账户并提交一个新链接即可。但，诸如这样的新链接在 Digg.com 上正以小时为单位如洪水般迅速地更新着，所以你很难收到实际效果，除非你发动几十个好友把你的新链接顶上排行榜。然而，就算你这样做了，也只能管用很短的时间。你也可以在其他的社交新闻类网站诸如 Yahoo Buzz (<http://buzz.yahoo.com>)和 Gamer Blips (<http://gamerblips.com>)发布你的链接，但缺点也同上。不过，如果你的链接能被一个游戏新闻网站发现并转到其他站点，这种手段就非常有效了。

## 使用 Twitter

Twitter 也是一个发布你的新游戏的有用工具。创建一个 Twitter 账户很简单，你还可以利用 Twitter 的一些小工具，比如让其他用户订阅你的发布内容。但要有效利用好 Twitter，你得做两件事情。



其一，设法让其他人订阅你的发布内容。就像在社交新闻类网站上你要做的那样，你一开始需要发动你的朋友和同事们来帮你推广你的游戏。吸引到一批你的游戏的有效订阅者是很耗时间和精力。靠发垃圾邮件和人工去堆这些有效链接也不是个办法。发展你的 Twitter 追随者们的最佳方法莫过于通过正当的途径了，通过这些途径把人们从 Twitter 吸引到你的站点去，并且开放人们通过你在 Twitter 上的发布内容注册你的网站账户的功能（[www.twitter.com](http://www.twitter.com) 提供了很多有用的工具和 API 来实现这些）。但是，一旦你搭建了有效的订阅群体，Twitter 也只能算你推广你的新游戏的几个最好的方式之一。你也可以使用诸如 Tweetmeme.com 这样的服务来帮你跟踪有多少人在 Twitter 上转发了你的游戏。

其二，使用 Twitter 的频道也是个好法子，它能让你在 Twitter 上的发布流传到原本到不了的地方（频道是用一个井字符号“#”标识的）。许多人会在 Twitter 上搜索他们喜欢的话题，而 Twitter 会对一些最受关注的频道进行实时更新。要让你的发布进入某频道，只需要在你的发布末尾加上“#flashgame”或者“#game”这样的频道标签就行了。Flash 游戏制作者和玩家都在关注着这一类频道——他们也刚好是你的游戏的受众。

## 使用 Facebook

使用 Facebook 网站来推广你的游戏的法子有难有易。最简单的就是发布一个链接就行了，只需要给 [www.facebook.com](http://www.facebook.com) 提交像这样的链接：

`http://www.facebook.com/share.php?u=\[url\]&t=\[title\]`

其中[url]是你的游戏的网址，[title]是你的文章标题的 HTML 格式化的字符串。虽然这法子很简单，但并不管用。这只能让你的 Facebook 好友看到你的游戏，而他们也许早就从你在 Twitter 上的发布内容中得到了这些重复的消息。

更好的法子是用 Facebook 的 API，但是这些 API 的细节不在这一章的讨论范围之内。你可以在 Facebook 的开发频道学到更多相关内容，网址是 <http://developers.facebook.com/>。

## 上传到游戏门户网站

另一个发布你的 Flash 游戏的方式是联系一些游戏门户网站并直接上传你的游戏。如果你能在那里展示你的游戏，就可以让它尽快为人所知。然而，效果不一定如你希望的那样持久。你可以上传你的游戏到如下两类游戏门户网站：社交游戏类网站和精品推荐类网站。

## 社交游戏类网站

诸如 Kongregate 和 Newgrounds 社交游戏类网站都允许你上传自己的游戏，并即时提供给它们的用户去玩。使用这些站点可以让你迅速得到关于你的游戏的反馈意见，还可以给你自己的站点赚来少许的反向链接。这些站点的最大好处在于，除了极少数内容不合适的东西（违法）之外，它们几乎对你提交的任何东西都照单全收。

然而，这些站点也有不好之处。首先，这样的网站都有各自的评级系统，就算你的游戏再优秀，也不一定能评上优。这些站点的用户群体的口味往往很刁（比如只喜欢僵尸保卫战这类的游戏），而你的游



戏不一定适合他们。因此，他们的评论和评分不一定对你的游戏有利。如果你懂投其所好（顺便说下，如果你的游戏选对了类型，占据了天时地利，也可能会得到这些用户的热捧），可以拿这些站点测试下自己的游戏是否能为大众所喜爱。一些站点也会定期举行竞赛并奖励一些现金给那些最受好评和被用户玩得最多的游戏的开发者们。但，这类竞赛往往是有隐患的，我们在下一节将会谈到这些。

## 精品推荐类网站

许多商业站点，如 MindJolt.com 和 AddictingGames.com 都允许开发者上传他们的作品，经站点审核后再发布出来。其中有些站点人气很高，但对作品的审核要求也颇为苛刻。这对开发者而言既是好事也是坏事。好就好在一旦你的游戏通过审核发布了出来，将有成千上万的人去玩它成千上万遍。如果该站点允许在游戏中植入广告，这意味着你短时间就会有可观的广告收益；而对于不允许嵌入广告的站点，比如 addictinggames.com 来说，你得不到钱（除非你能从中拉到赞助，这一点下一节将有解释），但你的站点也会收获到大量的访问量。坏就坏在，一旦你的作品被这些站点拒绝了，你会感到非常失败。这些站点可能出于很多种原因拒绝你的游戏的，这些原因也不总是和你的游戏的质量有关。精品推荐类网站会发布那些符合他们制定的特定分类的游戏，还会设法防止一些相似的游戏名称出现以至于淡化他们推荐的精品。而且，这些站点也在想方设法让尽量多的人回头重复玩尽量多的次数。因此，符合主流口味的游戏会比其他种类的游戏更受他们青睐。但这类的游戏也是非常多的。总而言之，想要让你的游戏在这些网站上榜，你需要用相当的技术实力做一个不错的游戏，除此之外还要一点运气。

在你上传你的作品到一个网站之前，要确保先阅读该站点的法律服务条款，以便知道你所要面对的一切。许多商业网站都声明自你一旦点击了上传按钮，它们就拥有了对你所上传的作品的合法所有权。要着重留意一下服务条款中的几组专业术语，确保你的作品适合上传给该网站：

Irrevocable

☐ ☐ Perpetual

☐ ☐ Exclusive

☐ ☐ Royalty-free

☐ ☐ Fully paid

☐ ☐ Sublicenseable and transferable license to use

☐ ☐ Ability to modify

☐ ☐ Prepare derivative works of

我们并不是说你不应该给上述站点上传你的作品；但一定要阅读这些法律条款，以了解清楚上传你的作品将有何后果。



## 靠你的 viral game 盈利

《发布一个 viral web game》一节阐述了推广作品的一种方式，这一切都由你独自完成是很难的。其实要想推广你的作品还有其他几种好的方法，也存在着一些理由会让你感觉根本不该发布出你的作品——至少不应该一开始就发布出来。

## 在游戏的页面中嵌入广告

靠你的游戏盈利的一种很普通的方式就是将它和 google 的广告放在同一个页面里，这样每天可以从这些页面的点击得到数百便士的收入。嵌入式广告在 google 上很容易得到，在 google (<https://www.google.com/adsense/>)注册一个账号，找一段 JavaScript 广告脚本，把它加入你的页面代码就行了。这些广告的形状大小都可以自定义，以便你能在页面中任意放置。在页面嵌入广告是你靠游戏盈利的最简单方式了。事实上，有些开发者更觉得它是能赚到钱的唯一方式。不过，要想靠在页面嵌入广告赚一大笔钱，你也需要大量的访问者，这很难实现。而且这种方式也有负面隐患。你可能喜欢把你的作品用大量广告围起来，这确实管用，但时间不会长，你的玩家们很快会对这些广告产生反感。正确的做法是，遴选少部分合适的位置来放广告。最后，你必须意识到除非你的游戏非常受欢迎，并且玩家只有上你的网站才能玩到，否则这些广告给你提供的收入是有限的，仅仅只占其他方式给你带来的盈利的一小部分。

## 参加游戏设计竞赛

当前网络上的游戏门户网站，甚至一些游戏新闻类和生活类网站上都刊有游戏设计竞赛的信息。对于这些网站而言，举办游戏设计竞赛可以让他们（往往）不花一个子儿就能获得一些游戏开发高手的作品。好吧，也许我说得过了点，但许多游戏设计竞赛的举办动机其实只能是让举办方获益。只有当可以保证公平，并且不需要开发者为了参与而过分的付出的游戏设计竞赛才是好的竞赛。你需要留意关于参与竞赛的那些印刷精美的法律协议。记得上节我讲门户网站中提到过的那些法律术语和条款吗？对于游戏设计竞赛而言也大同小异。大多数竞赛都是合法的，毕竟确实会提供奖金，也有人（甚至是你所熟知的高手）能赢得它们。但赢得这些奖金的几率不一定值得你参与（把你的游戏及源码交给它们）。所以小心为上，认真阅读竞赛的法律条款吧。

## 上传你的游戏到麻球网并嵌入广告

在麻球网，最常用的盈利方式就是将广告直接嵌入你的游戏中，嵌入式广告（前面提到过的嵌入网页的广告也一样）盈利的关键点在于 eCPM，即“有效的每千次展示费用”。eCPM 并不是盈利总量，而是显示你的游戏通过每千次点击能盈利多少？计算它的公式是：你的收入/游戏展示次数×1000。这个算法决定了你的 eCPM 值是会随着时间的变化而增减的，这取决于你放了哪些广告（不同的广告盈利率不同）以及你的游戏展示次数。eCPM 也就是广告商在收到每千次点击和展示就要付给你多少钱。因此，eCPM 值的高低可以显示你运营得好不好，而不是你盈利了多少。

Viral game 的开发者们会很快理解 eCPM 这个参数的存在意义，因为这个数值随着地域不同而变化很大。比方说一个 Viral game 在美国的游戏门户网上可以达到\$.50 的 eCPM（每千次可以得到 50 美分），但在中国或者南美洲的游戏门户网上可能就只有\$.01 eCPM。这是因为广告商左右着 eCPM 的值，他们更



愿意让自己的广告给那些可能会购买他们产品的人看到。由于绝大多数广告商来自于西方发达国家，因此这些地域的游戏门户网站会给你更高的 eCPM 值。但是，你也别低估那些非西方站点，这些网站的访客往往同样热爱玩游戏，他们会执着于把你的 Viral game 玩到通关，从而给你带来同样可观的 eCPM。除此之外，某些非西方站点也接受一些 eCPM 很高，但因其题材在西方暂时不甚流行，导致其在西方站点上不了正席的游戏。比如，我们可能会发现亚洲地区正流行着射击类游戏，与此同时，东欧地区却都在玩竞速游戏，而在北美，僵尸游戏正大行其道。总之，这些都没有规律，只需要你思路清晰。最好的方法就是紧盯着游戏门户网站，捕捉到流行趋势，然后投其所好。

在 Flash 游戏领域，最好的嵌入广告运营平台是由麻球多媒体提供的麻球广告 (Mochiads, <http://www.mochiads.com>)，此外诸如 CPMStar 这样的平台也不错，不过这里我们只针对麻球详细介绍。现如今，麻球广告对于 Flash 制作的 viral game 而言已经常见到不能再常见了。还有一个常识是：正如前所述的那样，你不必指望仅靠广告就能发财，而广告往往也不是你盈利的唯一来源。麻球媒体还提供了诸如使用情况跟踪、高分排行榜、加密、发行、版本化甚至是使用虚拟财产交易系统等多种盈利手段。我们在后面会接触到这些。和了解 Flash 游戏授权的常识一样，[www.mochiads.com](http://www.mochiads.com) 应该不是你作为 viral game 的开发者的第一站，你将在那里展开你的 Flash 游戏开发生涯。

接触麻球的第一步是进入麻球广告页面创建一个开发者账户，以这个账户上传一个游戏，就可以得到一段特殊代码，这段代码可以往你的游戏植入广告。大多数情形下，这段代码只有两三行的长度。待会我们仔细阐述麻球广告的游戏-广告整合系统。除了植入广告，麻球媒体还额外提供了一些服务。麻球高分榜 (Mochi High Scores) 允许你在游戏中添加一个开放式的高分排行系统。麻球发布服务 (Mochi Distribution) 允许你通过麻球网络 (Mochi Network) 向麻球网的一些门户游戏网站的合作伙伴发布游戏，你可以通过这个系统同步更新你的游戏，而不必每次重新上传新版本给这些网站，同时还允许你对你的游戏源码加密。

最后，麻球媒体最近开放了一个虚拟物品交易系统：Mochicoins (Mochicoins)。麻球虚拟物品交易 (Microtransaction) 也是赚钱的一种方式，它允许玩家之间交易游戏中的虚拟财富 (新关卡的授权，新武器，等)。由于麻球广告几乎不排斥任何游戏，要挤进 Mochicoins 也就变得压力很大了。不过，只要你的游戏确实品质优秀，而你运气够好地加入了 Mochicoins 系统，麻球媒体将动用可观的空间资源替你推销。

## 申请游戏授权，获得赞助

前面花了很多篇幅来教你怎样将你的游戏通过游戏门户网站摆在玩家的屏幕前。不过，某些情况下，你不应该马上做这些。为啥呢？因为品质优秀的游戏目前的需求量很高。与此同时，对于游戏门户网站而言，新游戏是关键点，它们总需要提供新游戏来保证访客的回头率。

如果你的游戏是新出炉的，而且品质优秀，你的手头上就有了它们所追寻的东西。但若你立马就传到了社交游戏类网站或是麻球广告上，你就白白失去了这一优势。如果你发现你的游戏有做大的潜力，你首先要做的就是出售游戏授权以及赚取赞助。而一旦曝光了 (不管通过什么渠道公布出去了)，对游戏的授权盈利和赚取赞助都是致命的。富足的游戏门户网需要独家的内容，会愿意花钱买你的独家授权。这意味着，要赚最多的钱的话，本章前半部分你都不必看了，看这几节就行。然而，要是游戏授权和赞助那么容易得



到的话，前面也没必要写那么多了，而你也应该早就发财了。事实上，申请游戏授权和获得赞助更像一个越来越艰苦的战场。每天都有优秀的游戏参与其中的竞争，所以，你最好的机会就是做一个极佳的游戏以便能吸引住赞助者的眼球。你可以尝试获取以下几种类型的授权和赞助。

## 独家授权

独家授权指的是游戏的使用权被独家买断的情形。这种授权方式可以附带或者不带源码，但必须只能在特定的一家网站上展示，有时也会限时。大多数情况下，这是游戏开发者实现最大盈利的方式。

## 赞助金

赞助金意即游戏门户网站（或者它们的赞助方）付你的钱，以便在你的游戏中植入他们的广告（比如要你先换下麻球广告），然后会不遗余力的推广你的游戏以便保证广告效果。有些情形还会以游戏的被点击次数给你分红。总的来说，游戏开发者能从赞助金的方式赚的钱几乎可以和独家授权一样多。

## 非独家授权

非独家授权（通常来说）赚不到独家授权和赞助金那么多的钱，但给游戏开发者留的余地也大。非独家意味着赞助方和被授权方要付你钱放上他们的广告，否则其他广告商也会放的。这意味着游戏开发者虽然不能从一个游戏门户网站大赚一笔，但是可以同时从多家站点一起赚到一个可观的总额。

## API 授权

API 授权很容易通过审核，但赚不了很多。部分游戏门户网站会付给开发者一笔小钱（比方说 50 美元），以将他们的高分排行榜加入你的游戏中，或者把你的游戏整合到他们的游戏里。整理自己的 API 代码很容易，通过这种途径可以很快赚一笔钱，只是数目有限。

## 给游戏拉赞助

现如今有两种基本的方法可以让你的游戏通过授权审核或者拉到一些赞助。第一种即你挑选一家合意的游戏门户网站，给该网站致函。如果你与该站站长有些交情，或者你本身有制作优秀游戏的口碑，以这种方式会容易许多。但也有一些案例显示有新手直接联系游戏门户网站而赚到了不错的赞助或者授权收入。然而，第一印象是很重要的，如果你要联系这些站长忙人，一定要确保你手上的货是精品——即优秀的，好玩的高品质游戏。



## 使用 FlashGameLicense.com 网站

第二种较普遍的方法就是使用 FlashGameLicense.com 网站，该站面向游戏开发者提供一些麻球媒体的补充功能。该站建于 2007 年，起初只是作为游戏开发者们公开作品前的上传地址，以方便赞助者和游戏门户站长可以浏览到这些作品并为之竞价。不久后它为开发者们添加了论坛版块用以提供反馈信息，以及一个游戏商店用以出售那些不准备用来拉赞助的作品。最近，该站还添加了发行分站 FlashGameDistribution.com(<http://flashgamedistribution.com/>)，以便让开发者发布自己的作品。而 GamerSafe.com 则为 Flash 游戏提供了随玩随存档的功能，以方便虚拟道具交易。要想加入 [www.flashgamelicense.com](http://www.flashgamelicense.com) 很简单：创建账户然后上传游戏就行了。FlashGameLicense.com 与麻球媒体的不同之处在于它是一个私密性质的网站。所有上传上去的游戏都被上了站点锁和加了密。这为开发者给赞助商们展示作品提供了一个良好的环境，保证了作品基本不会泄露出去。作为开发者，你可以设定你所接受的授权类型及作品的版权价格。FlashGameLicense.com 会抽取 10% 的利润，但若你能找到一家好的赞助，这点付出是值得的。

## 使用 FlashGameDistribution 及 GamerSafe 网站

倘若你的作品没有获得授权审核和赞助，FlashGameLicense.com 还会利用 FlashGameDistribution.com 分站将你的游戏展示给网站的用户。这一服务确实算得上锦上添花，但 FlashGameLicense.com 还有更有趣的一招：GamerSafe.com。GamerSafe.com 给开发者们提供了三项非常使用的功能。其一，它拥有游戏的线上存档机制，这使玩家可以随玩随存档。对于关卡数较多的游戏，这一功能可谓无价之宝。这几年来，Flash 游戏的数据保存地址都在本机，这只能方便那些一直用同一台电脑进行游戏的玩家们。GamerSafe.com 的这一功能（麻球网现在也有类似功能了）大大放开了线上 Flash 游戏的类型限制：RPG 游戏，历时较长的探险或战争类游戏等都行了。GamerSafe.com 也提供了植入游戏的交易系统，可以像麻球网那样实现虚拟物品的交易。值得一提的是麻球网最近开始添加一种新功能，可以匹配 GamerSafe 上的“相同的游戏”，以及另一项名为 HeyZap([heyzap.com](http://heyzap.com))的新业务也在快步跟进。截止本书出版为止，HeyZap 已经添加了植入游戏的广告功能。正如你所想的那样，这将是一个竞争激烈的斗场，因此最好尽快入驻其中，并尽可能多地了解行业的最新动态和趋势。

## 使用 Adobe Flash Platform Services

2009 年的晚些时候，Adobe 宣称将进入 Flash 的发布应用的商业领域，并提供了三项非常有趣的服务：Distribution，Social，以及 Shibuya，Flash 游戏开发者可能会觉得它们很有用。

Distribution 服务是 Adobe 对麻球媒体在相关领域的回击。在与 Gigya 的合作下，Adobe 发布了 Share menu，它包含了你可能需要的所有应用。Share menu 开放了对 Adobe 相关产品的广告投放，使用情况跟踪，以及推广功能。其中的推广功能似乎是一项收费服务，而广告投放功能是用户可以自选的，并可以与其他广告服务（如麻球）一起使用。使用情况跟踪功能与麻球网的相关服务类似，不过似乎因为 Adobe



收购了 Omniture——世界上领先的网络分析公司之一——而显得很给力。你可以[点击这里](#)以进一步了解 Adobe 的 Distribution 服务细节：

<http://www.adobe.com/devnet/flashplatform/services/distribution/>

Social 服务提供一种 API，可以实现多种社交网络的数据共享和验证。目前该 API 支持 Facebook，MySpace，Twitter，雅虎，谷歌和 AOL。你可以[从这里](#)下载 Adobe 的 Social 服务条款：

<http://labs.adobe.com/downloads/social.html>

Shibuya 服务（目前仍在测）是 Adobe 的一个为基于 AIR 的产品的发布和交易平台。Adobe AIR 是一种可以将 Flash 应用转换为当前流行的操作系统（如 Windows，Mac OS）桌面应用程序的技术。Shibuya 允许用户对这些 AIR 应用先试用再购买使用权。你可以在[这里](#)找到关于 Adobe Flash Platform Services 的更详细的介绍：

<http://www.adobe.com/devnet/flashplatform/services/>

## 保护好你的 viral games

在我们告别 viral games 的应用话题而开始接触一些枯燥的代码之前，还有两个方面的内容需要阐述：站点锁和加密。

## 使用站点锁

我们先前说过，站点锁是一个工具，它让游戏开发者得以只允许自己的游戏在某个特定的游戏门户网站上才能玩到。不过本书的 game framework 里没有提供站点锁的代码。这里有一段可生效的代码样本：

```
var validDomain:String="8bitrocket.com";
var isValidDomain:Boolean=false;
var currentDomain:String = this.root.loaderInfo.url.split("/")[2];
trace("currentDomain=" + currentDomain);
if (currentDomain.indexOf(validDomainString) == (currentDomain.length - _
validDomainString.length)) {
    isValidDomain=true;
}else{
    isValidDomain=false;
    systemInvalidDomain();
}
private function systemInvalidDomain():void {
```



```
navigateToURL(new URLRequest("http://www.8bitrocket.com"), 'newwindow');  
}
```

以上代码的基本思路是：使用 `this.loaderInfo.url.toLowerCase` 检查有效域名（“8bitrocket.com”）。如果检查不到，则调用 `systemInvalidDomain` 函数，该函数可以让浏览器转链到某个特定的网站。如果某个非授权网站试图盗用你的游戏，当有人访问该游戏时页面就会直接转链。

## 给你的游戏加密

由于麻球和 FlashGameLicense.com 为 Flash 游戏提供的加密能力都很有限，要做到最安全的话，你得自己另外给你的 SWF 文件加密。为什么要这样做？因为有些人会破解你的游戏源码，去掉其中的广告，赞助内容 预加载画面以及信贷信息 从而将游戏据为己用。我们发现 SWF 加密的最好方法是使用 Amayeta SWF Encrypt(<http://www.amayeta.com/>)。该应用会将 SWF 文件中的代码转成乱码，同时还会加入钩子函数阻止一般的 SWF 反编译器工作。缺点是会增加 SWF 文件的体积，有时会增加两倍甚至更多，取决于你选择的加密强度等级。虽然该服务也不免费，但其价格（约 150 美元）值得你花每一个子儿，尤其是当你用它发现了一些代码正试图破解你的源码之时。顺便提一句，麻球网也会在加载你的游戏时也进行一定的加密，但即使如此麻球网也仍然推荐你另外采取加密手段。

## 营销你的 viral games

你的 viral game 的营销战略取决于你自己做游戏的目的。最普遍的两个目的莫过于赚钱和出名。这里我们来探讨一下如何采取相应的营销手段去实现这两个目的。首先，如果你的目的是赚更多的钱，那么你应该遵循以下步骤：

- 1、用站点锁锁上你的游戏并放在你自己站点的隐藏目录下。不要往游戏里植入广告。

- 2、以私人身份将游戏提交至一些商业游戏网站（Addicting Games, Big Fish 等等），这些网站可能赞助你的游戏。如果这些站点不允许上传，就将你的作品的带锁版本用电子邮件发给它们。

- 3、敬候佳音……（译者：\_-!）

- 4、如果你没有因此拿到合同，就将你的游戏提交到 FlashGameLicense.com。记得标识你的游戏并未发行，而是接受赞助中。

- 5、继续敬候佳音至少三周，除非你拿到了想要的赞助。

- 6、参见如何出名的步骤。

其次，如果你的目的是尽可能地出名，试试以下步骤：

- 1、在你的私人站点上创建一个页面。

- 2、将你的游戏上传至麻球网，并在麻球发布网上添加广告和跳转链接。

- 3、将你的游戏提交至某大型商业游戏门户网站，该网站应不支持广告（比如 AddictingGames.com）。你这样做赚不到钱，但一旦你的游戏被发布出来，你会收获一些反向链接。要确



保的是你提交的游戏版本应该是不带广告的。使用麻球广告过滤工具过滤掉这一域名，以不让广告在这一站点的游戏中显示出来。

4、在你的网站上制作一个页面，放入你的游戏，并加入麻球的<embed>广告代码，因此当其他游戏门户网循踪找到你的游戏时，就会展示出带广告的游戏版本。

5、将你的游戏提交至你能找到的任何社交游戏类或精品推荐类网站。别忘了要提交给 FlashGameDistribution.com。

## 利用其他主要的网络资源

紧跟行业潮流不掉队是很重要的，以下这些站点可以给你带来 Flash 游戏行业的最新资讯：

游戏类网站：

Adobe Flash Game Technology Center: <http://adobe.com/devnet/games/>  
Gaming Your Way: [www.gamingyourway.com](http://www.gamingyourway.com)  
Flash Game Blogs: [www.flashgameblogs.com](http://www.flashgameblogs.com)  
Photon Storm: [www.photonstorm.com](http://www.photonstorm.com)  
Ickydime: [blog.ickydime.com](http://blog.ickydime.com)  
Game Poetry: [www.gamepoetry.com/blog/](http://www.gamepoetry.com/blog/)  
Blog.sokay.net: [blog.sokay.net](http://blog.sokay.net)  
Jobe Maker: [jobemakar.blogspot.com/](http://jobemakar.blogspot.com/)  
Urban Squall: [www.urbansquall.com/blog/](http://www.urbansquall.com/blog/)  
Michael J. Williams: [gamedev.michaeljameswilliams.com](http://gamedev.michaeljameswilliams.com)  
Freelance Flash Games: [www.freelanceflashgames.com](http://www.freelanceflashgames.com)  
Iain Lobb: [blog.iainlobb.com/](http://blog.iainlobb.com/) (site belongs to the technical editor of this book)  
8-bit Rocket: [www.8bitrocket.com](http://www.8bitrocket.com) (our site)

商业博客类网站：

Mochiland: [mochiland.com](http://mochiland.com)

Flash 相关类网站：

Flash Enabled: [www.flashenabledblog.com](http://www.flashenabledblog.com)  
ActionScript.org: [www.actionscript.org/](http://www.actionscript.org/)  
Flash Kit: [www.flashkit.com/](http://www.flashkit.com/)  
Scott Jeppesen: [scottjeppesen.blogspot.com/](http://scottjeppesen.blogspot.com/)  
Flash Focus: [www.flashfocus.nl/](http://www.flashfocus.nl/)  
Kirupa: [www.kirupa.com](http://www.kirupa.com)



## 准备创建隧道惊魂游戏

游戏中除了一些加入的第三方代码，比如预加载代码，麻球媒体服务框架之外，其余代码我们都将放在 com.egf.framework 包内。不管你是使用 Flex, Flash Develop, Flash Builder, 还是 the Flash IDE，你都必须为你的项目创建一个文件夹来放置所有的代码文件。

就像这本书中所有的游戏一样，隧道惊魂也使用第二章中介绍的游戏框架。我们要首先在 Flash 和 Flash Develop 中分别创建该游戏必须要用到的文件结构。

## 在 Flash IDE 中创建游戏项目

现如今，下面的创建步骤你应该非常熟悉了。

打开 Flash,我用的是 CS3,不过 CS4 和 CS5 版本的用起来是一样的。

在/source/projects/tunnelpanic/flashIDE/文件夹下创建一个名为 tunnelpanic fla 的文件。

在/source/projects/tunnelpanic/flashIDE 文件夹下，创建下面的目录树（包结构）：  
com/efg/games/tunnelpanic/。

设置帧速率为 30。窗口的高为 400，宽为 600

将文档类指向 com.egf.games.tunnelpanic.Main。

这个文档类我们将会在这章的稍后部分创建。

现在将添加我们要用到的第三方代码。在发布设置中选择 Flash ➤ ActionScript 3 设置。

点击浏览路径按钮。并且找到我们在第二章中创建的/source 文件夹。

选择 classes 文件夹，然后点击选择按钮。现在 com.egf.framework 包就可以在我们的游戏中使用了。

## 在 Flash Develop 中创建游戏项目

以下是在 Flash Develop 中创建游戏项目的步骤：

1、在/source/projects/tunnelpanic/的路径下创建一个名为 flexSDK 的文件夹（如果你以前没做过这些话）。

2、启动 Flash Develop 并创建一个新项目；选取 Flex 3 项目类型，并命名为隧道惊魂。Location 设置为/source/projects/tunnelpanic/flexSDK 文件夹，package 设置为 com.egf.games.tunnelpanic。注意别选中 Create Folder For Project 的选取框，不要让 Flash Develop 创建其缺省的项目文件夹。最后点 OK 按钮。

3、现在需要给项目框架添加类路径了，点选 Project ➤ Properties ➤ Classpaths menu item。



4、点击 Add Class Path 按钮，提交我们之前创建的/source 文件夹。

5、点击 OK 按钮和 Apply 按钮。

6、接下来更改舞台大小和帧频。再次点选 Project > Properties > Classpaths menu item，将帧频设为 30，舞台宽高各 600 和 400。

现在游戏项目的基本构架已经做完了，我们将进一步探讨一些关于构架类的内容，并构建一个可重用的架构体系。

(对于 Flex Builder、Flash Builder 以及其他 IDE，请根据我们之前提供的文档创建项目并设置默认设置。在 Flash Develop / Flash IDE 协作的工作流程中，我们常用 Flash 来组织素材，用 Flash Builder 编写代码。如果你也是这样做的，就应该采用我们使用 Flash IDE 的包结构继续而不是 Flex SDK 的结构)

以下是采用 Flash IDE 所构建的文件夹结构：

```
/source/projects/tunnelpanic/flashIDE/com/efg/games/tunnelpanic/
```

以及 Flex SDK (使用 Flash Develop) 的：

```
/source/projects/tunnelpanic/flexSDK/
```

```
assets/
```

```
bin/
```

```
obj/
```

```
lib/
```

## 实现在 Flex SDK 中的预加载

在 Flex SDK 和 Flash develop 中实现预加载相对来说较轻松，但需要你从根本上更改应用结构。基于 Flex SDK 的应用是没有 Flash 的时间轴的，所以素材和代码都需要在该应用运行之前加载。这一点和 Flash IDE 的应用很不同，后者的时间轴可以让我们很方便的从第一帧移除素材从而顺理成章地实现加载。

如果使用 Flex SDK，我们需要创建一个名为 Preloader.as 的类，将之放在我们的项目文件夹中，与 Main.as 同级。本章之前我们已经在 Flash Develop 中创造过了一个预加载器，如果你点选 New Project > AS3 Project 中的 Preloader 选项就可以使用。但在这个例子中我们还是再自己详细地过一遍以便大家熟悉详细过程。

## 添加编译器指令

为正确预加载隧道惊魂的 Main.as 的 Flex SDK 版本，我们先得了解一下编译器处理 Main.as 类的过程。首先，它不再是文档类 (或 Flash Develop 中的必被编译类)，取而代之的是 Preloader.as。这样



做是为了使用 Preloader 来加载 Main，要实现这一点必须让编译器误认为 Main.as 起始于 SWF 文件的第二帧。而第二帧其实并不存在，我们怎么做到这些的呢？

在 Flash Develop 中，保持项目为打开状态，进入 Project > Properties > Compiler Options 菜单，你必须在 Compiler Options 中添加一项新的编译选项如下：

```
-frame start com.efg.games.tunnelpanic.Main
```

这条指明告诉编译器：一个名为 Main 的类将在第二帧生效。如图 12-1：

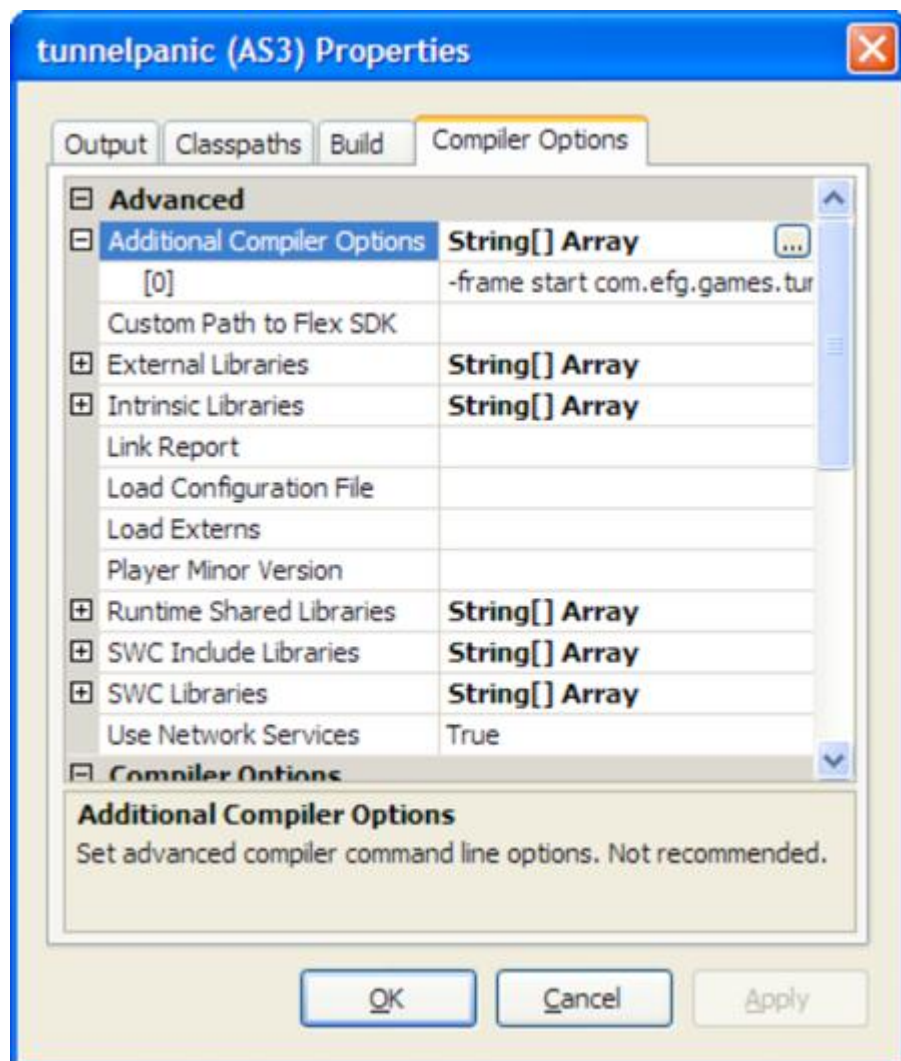


图12-1 Flash Develop的编译器选项设置

## 添加预加载代码

接下来，我们需要创建一个名为 Preloader.as 的新类如下：

```
package com.efg.games.tunnelpanic  
{
```



```
//Must set compiler option of "-frame start Main" in Additional compiler options
//for this to work.
//not needed with Mochi pre-loader ad
import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.display.DisplayObject;
import flash.display.MovieClip;
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.text.TextField;
import flash.text.TextFormat;
import flash.utils.getDefinitionByName;
public class Preloader extends MovieClip
{
    private var appBackBD:BitmapData = new BitmapData(600, 400, false, 0x000000);
    private var appBackBitmap:Bitmap = new Bitmap(appBackBD);
    private var textfield:TextField = new TextField();
    private var headerTextfield:TextField = new TextField();
    private var textFormat:TextFormat = new TextFormat("_sans", "11", "0xffffffff", "true");
    private var loadingString:String;
    public function Preloader()
    {
        trace("pre loader");
        textfield.defaultTextFormat = textFormat;
        headerTextfield.defaultTextFormat = textFormat;
        headerTextfield.text = "loader on screen";
        textfield.x = 280;
        textfield.y = 200;
```



```
addChild(appBackBitmap);
addChild(textfield);
addChild(headerTextfield);
addEventListener(Event.ENTER_FRAME, checkFrame);
loaderInfo.addEventListener(ProgressEvent.PROGRESS, progress);
}

private function progress(e:ProgressEvent):void
{
    // update loader
    trace("loader");
    trace(e.bytesLoaded + "/" + e.bytesTotal);
    var loadingInt:int = (e.bytesLoaded / e.bytesTotal) * 100;
    loadingString = "Loading: " + loadingInt + "%";
    textfield.text = loadingString;
}

private function checkFrame(e:Event):void
{
    if (currentFrame == totalFrames)
    {
        removeEventListener(Event.ENTER_FRAME, checkFrame);
        startup();
    }
}

private function startup():void
{
    // hide loader
    removeChild(appBackBitmap);
```



```
removeChild(textfield);

stop();

loaderInfo.removeEventListener(ProgressEvent.PROGRESS, progress);

var mainClass:Class = getDefinitionByName("com.efg.games.tunnelpanic.Main") as Class;
addChild(new mainClass() as DisplayObject);
}
}
}
```

可见预加载的实现是很简单的。我们添加了一个 BitmapData 背景(appBackBitmap 中的 appBackBD) 和一个 TextField(textfield)用以显示加载状态信息，同时创建了两个事件侦听函数：checkFrame 用以侦听 Event.EnterFrame 事件，progress 用以侦听 LoaderInfo 类中的 ProgressEvent.PROGRESS。

progress 函数只负责更新 textField 中显示的加载百分比，而 checkFrame 则等候第一帧加载完成，播放指针移到之前被编译器创建的第二帧的那一刻。一旦 checkFrame 检测到已经是第二帧 (currentFrame == totalFrames)，就会调用 startup 函数。

startup 函数将 appBackBitmap 和 textField 移除出显示列表，并在显示列表添加一个 Main 类的实例，为此我们还要对 Main 的构造器和 init 函数稍事修改，使之可以生效。

最后，我们通过创建的这个 Main 类的实例对象将我们的这些代码与先前的代码整合了起来，并添加到了舞台的显示列表中。我们这样实现的：

```
var mainClass:Class = getDefinitionByName("com.efg.games.tunnelpanic.Main") as Class;
addChild(new mainClass() as DisplayObject);
```

现在，你已经学会了使用第二个类(Preloader)和特殊的编译器指令 ( -frame start com.efg.games.tunnelpanic.Main )来加载 Main.as 和其他素材了吧。

## 使用 Flash IDE 进行预加载

在 Flash IDE 中实现预加载与 Flex SDK 不同。主要区别在于时间轴和库与预加载进程的互动上。在以下例子中，我们将使用本章游戏隧道惊魂所需要的素材。先在库中导入三个声音文件来让游戏的音效更具个性。如果你从本书的网站上下载了这些素材，可以跟着我做。若没有，你就要去找三个声音剪辑文件了：一段标题音乐，一段游戏进行时音乐和一段飞船坠毁的音乐。



## 往库里添加文件

你必须将这三段音乐添加进库中,如图 12-2 所示。你也许注意到库里还有一个名为 SoundHolder 的 MovieClip,我们待会再来管它。

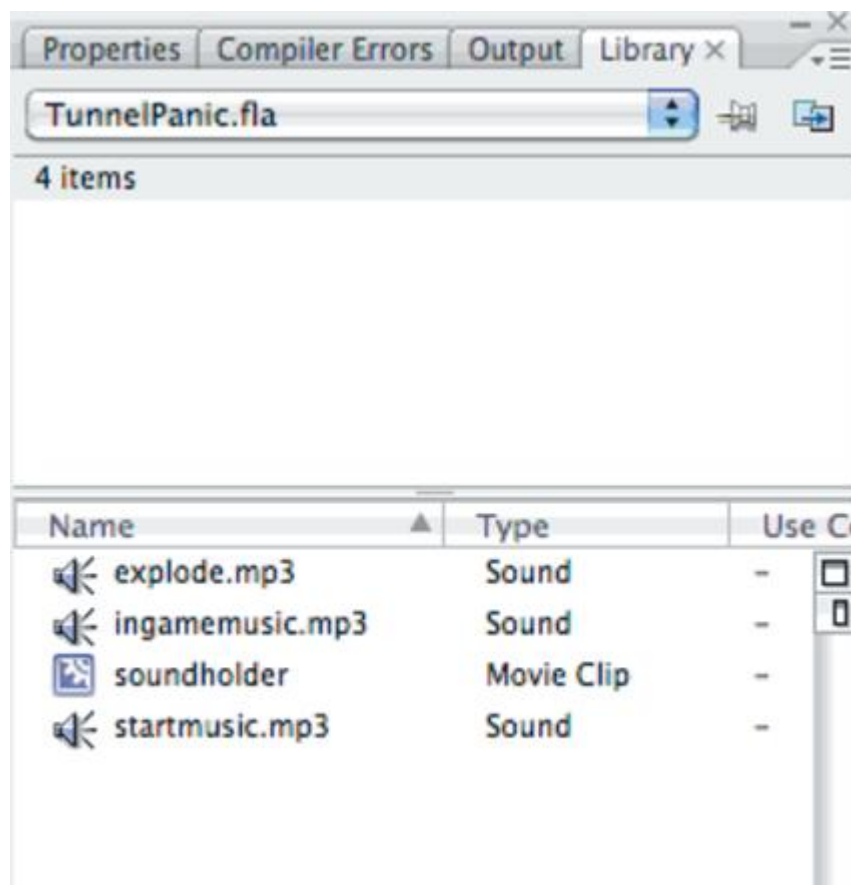


图12-2 隧道惊魂游戏的库和预加载内容

## 创建时间轴内容

本游戏的 FLA 文件的时间轴包含两个层,每层各含三个帧。第一帧除了一条动作面板的 stop(); 指令外没有其他内容。第二帧包含一个 soundHolder 的实例,我们下一节就教你如何添加这个实例。第三帧和第一帧一样,仅包含一条动作面板的 stop(); 指令。

如图 12-3 是示例。

时间轴的两个层分别名为 code 和 assets,code 层包含第一帧和第三帧的代码。Assets 层在第二帧容有 soundHolder 的实例,如图 12-3:

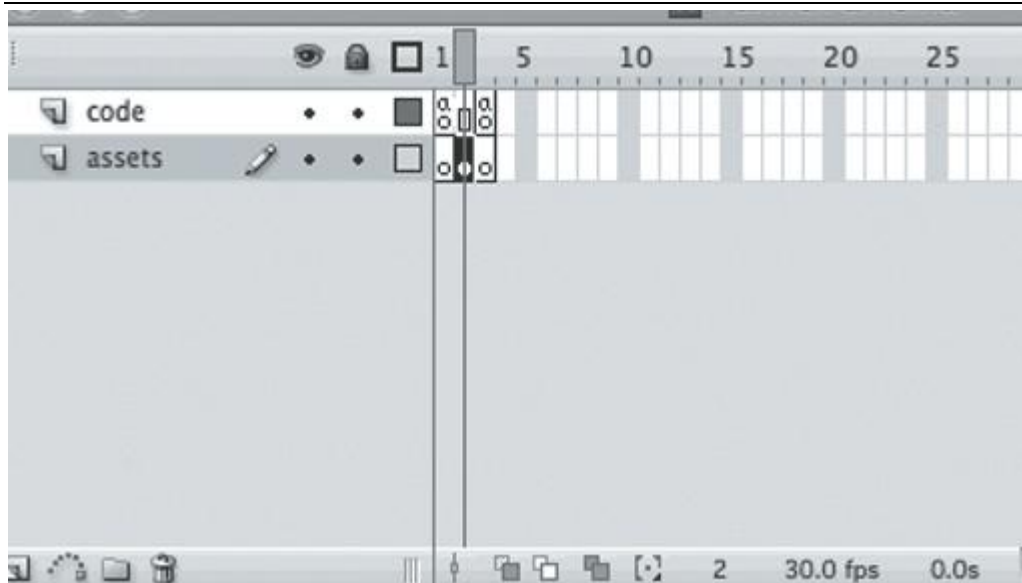


图12-3 隧道惊魂游戏的时间轴和预加载内容

## 创建一个影片剪辑作为素材的容器

我们还需要一个影片剪辑作为第二帧的素材的容器。姑且称之为素材容器。使用该容器可以帮助加载器加载每一个游戏需要的元素，而不必给第一帧造成过重的负担。如果素材都输出到第一帧的话，可能会有错误，因为 Main.as 在所有素材加载完成之前是不会运行的。这与我们需要提供预加载信息的目的背道而驰，因为如果素材没有加载完成，预加载信息无法得以显示。在 Main.as 运行之前加载所有素材会造成极大延时，此时我们只能看到屏幕上空空的一片。我们之前在库里创建的 soundHolder 就是用来容纳本游戏的所有声音素材的。如果还有图片素材需要加载，我们就要另外创建一个 graphicsHolder，或者干脆把它们也放在 soundHolder 里。如果你确实要用到混合类型的素材，你可能希望把该元件重命名为 assetsHolder 而不是 soundHolder。随便改吧，因为这个名字不会在代码里用到的。

素材容器（即我们的这个例子中的 SoundHolder）包含了游戏所需的全部素材，等待被代码从库中实例化出来。如果素材已经存在于你的主时间轴上，或者位于另一个影片剪辑中，就不需要将它们放入 SoundHolder（或是 assetsHolder）中了。本书的游戏都不会用到太多的主时间轴素材，因此为了预加载，把你的所有素材都放入一个素材容器中吧。

## 链接素材

现在库中有三段声音剪辑，必须为它们设置链接名，但不能在第一帧设置。因为链接名必须映射我们在 Main.as 中的初始化对象名。图 12-4 显示了 explode.mp3 这段音乐在库中的链接设置。注意到链接名称(SoundExplode)和库里的素材名(explode.mp3)是不同的。你可以在导入完成后将这些名称改为一

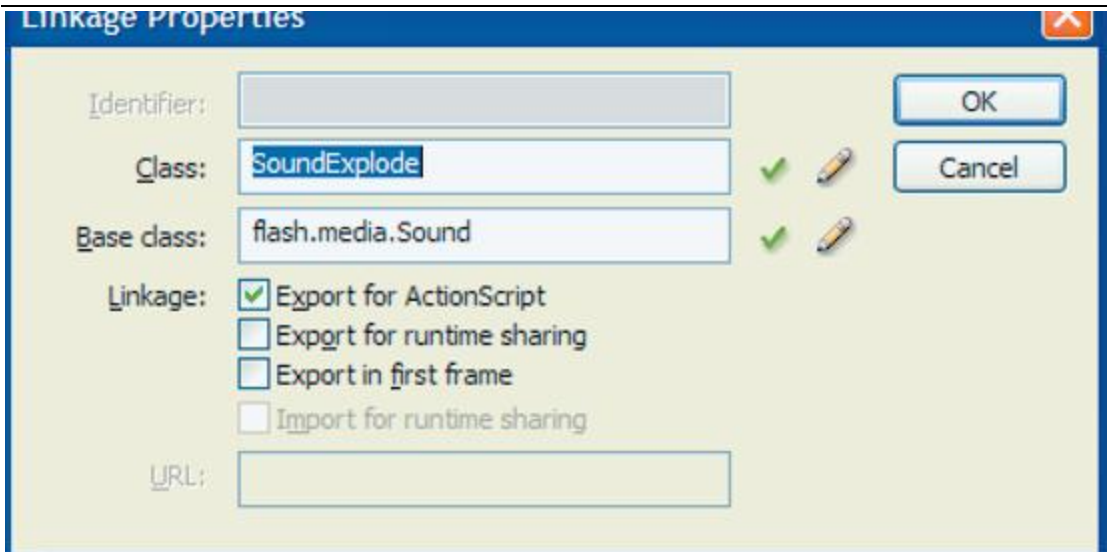


图12-4 explode.mp3的链接设置

表12-1 是库中三段声音剪辑各自的链接类名的图示

Library	Class
explode.mp3	SoundExplode
ingamemusic.mp3	SoundMusicInGame
startmusic.mp3	SoundMusicTitle

图 12-5 是整个库的图示

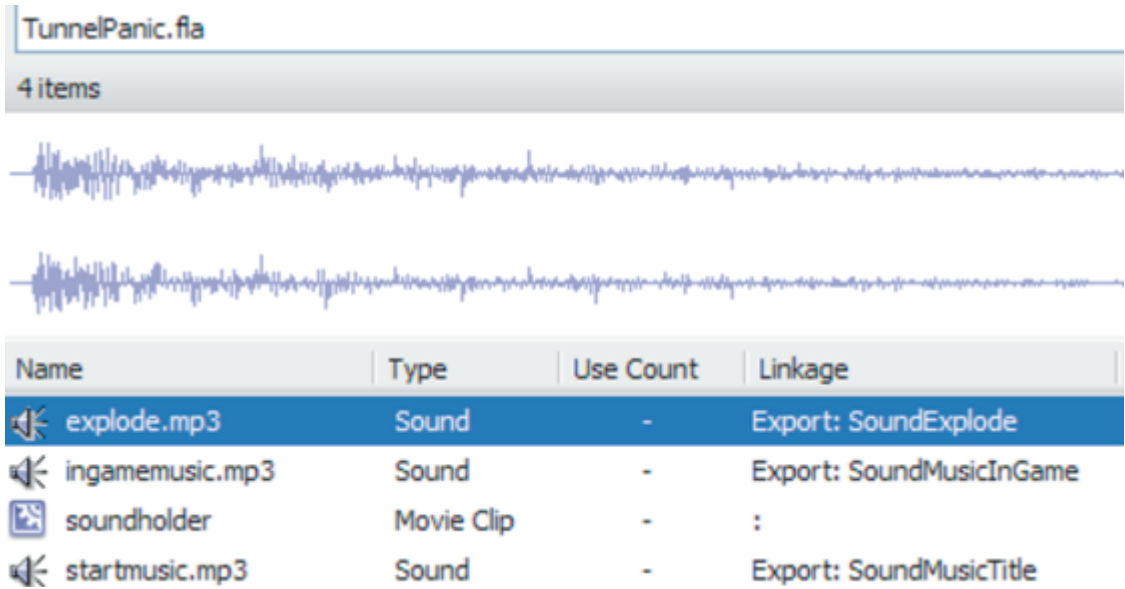


图12-5 库中素材的所有链接属性

## 将素材放入素材容器中

接下来要创建素材容器，由于本游戏只有声音素材，我们可以把素材容器命名为 soundHolder，这个名字可以随便取。在该容器中添加如图 12-6 所示的时间轴：

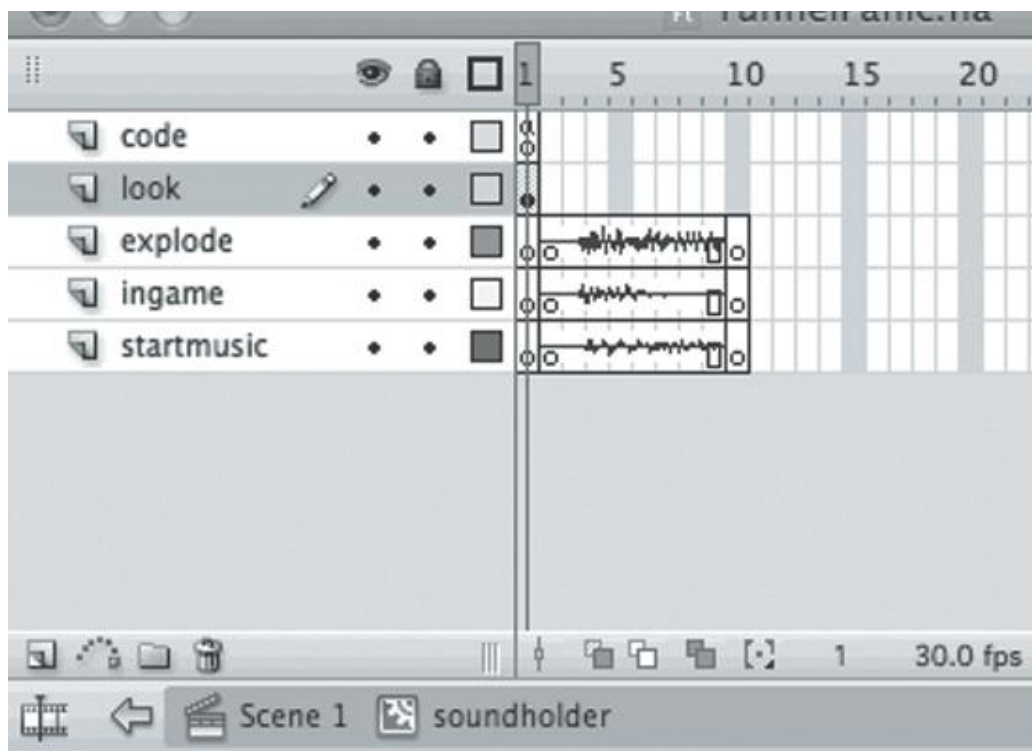


图12-6 soundHolder剪辑

soundHolder 含有 10 帧，但只有其中两帧是必不可少的。其余的几帧则是为了达到可视化的目的。另外还含有 5 个层，第一层（code 层）的第一帧只有一条动作面板的 stop 指令，第二层则含有你需要放入的素材。在这里放一个灰色的方形并用红字写上 sounds，这样做事为了实现可视化。所有的声音素材将在其各自所处的层里从第二帧开始播放。所以第一帧将会显示一片空白，除非你往 look 层另加了可视化元素。所有的声音素材都被放在其各自所处的层里的第二帧，把每一个声音单独放在时间轴的一个层里播放是必须的。为什么所有声音要从第二帧开始播放？原因很简单：如果它们都从第一帧开始放的话，你就会有在播放指针移往第二帧时听见它们的播放效果。如果还有其他可视化素材要预加载，把它们都放在一个层里没错，但每个素材单独放一个层的结构更便于管理。这种结构给后期添加和删除素材提供了便利。

## 将素材容器放入时间轴

最后要做的就是将 soundHolder 的一个实例放入 FLA 文件主时间轴的 assets 层的第二帧了，如图 12-7：

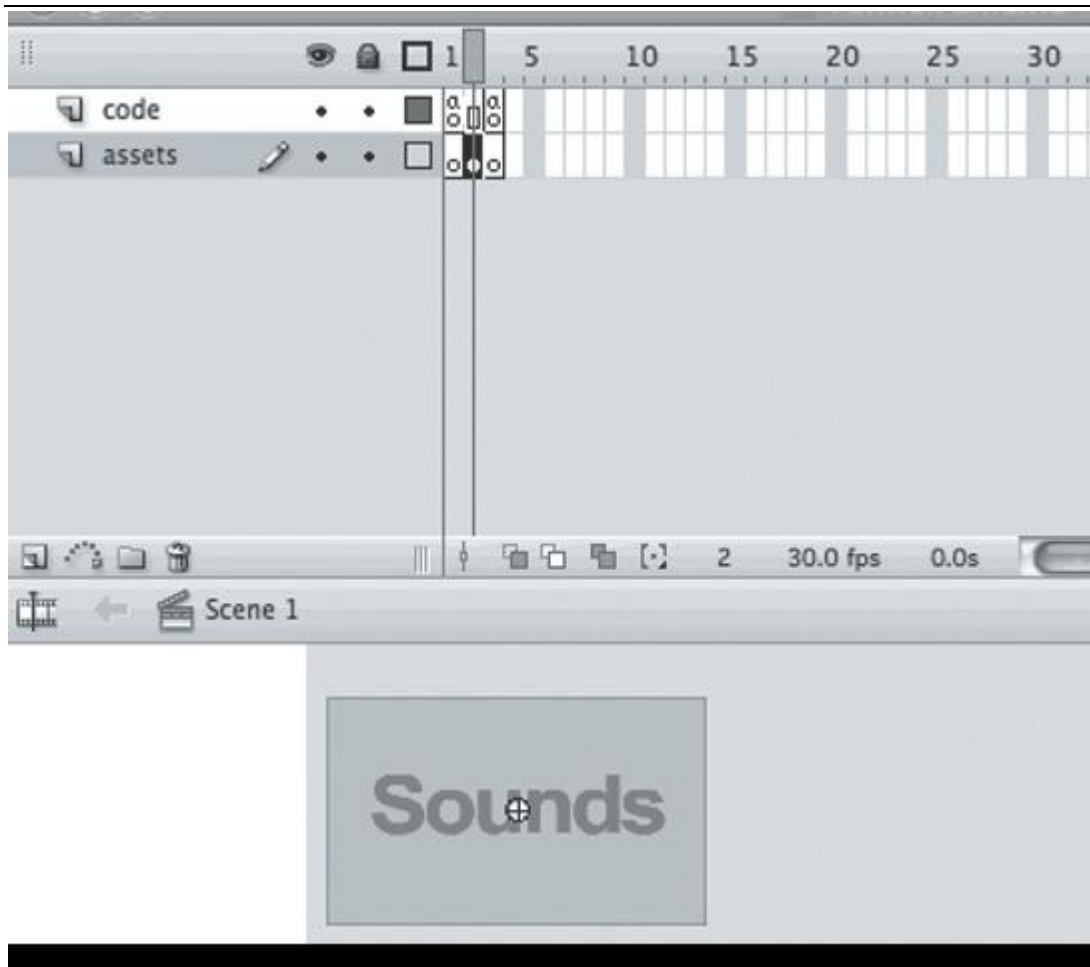


图12-7 soundHolder实例在主时间轴上的摆放

## 对于 Flash IDE 的预加载所须做出的框架更改

为实现在 Flash IDE 中向 Main.as 预加载相关内容，必须对 com.efg.framework 和 Main.as 做重要修改。不像 Flex SDK 的预加载器，基于 Flash IDE 的预加载器不需要 Preloader 类。我们必须给 Main.as 的状态机制中添加一个新状态，但这些内容对于 Flex SDK 就没有必要了。

## 向基于 Flash IDE 的 framework 中添加预加载状态

为实现基于 Flash IDE 的预加载，需要对 com.efg.framework 下的类加以修改，Flex SDK 的使用者也要跟着看看，因为其中的有些改动也适用于在 Flex SDK 的开发环境下。

首先，需要添加一些新状态，我们需要增加一些新的预加载常量到 FrameworkStates.as 的变量定义区中。另外还要为麻球广告和高分排行添加常量，不过我们放在下一节讲。将以下代码添加到 FrameworkStates.as 的变量定义区中，要注意 Flex SDK 的使用者不需要使用 STATE\_SYSTEM\_PRELOAD 状态，但仍然需要添加这一状态，以防今后会用到 Flash IDE 下的 framework。



```
public static const STATE_SYSTEM_PRELOAD:int = 11;

public static const STATE_SYSTEM_MOCHI_HIGHSCORES:int = 12;

public static const STATE_SYSTEM_PAUSE:int = 99;
```

## 添加预加载状态下所需的变量

现在往基于 Flash IDE 的 com.efg.framework.GameFrameWork.as 里的预加载器添加必要的变量。它们分别是一个 Boolean 类型的 preloaderStarted 以及一个 String 类型的 preloadString。预加载状态的函数将会使用它们为用户提供可视化文本。

```
public var preloaderStarted:Boolean=false;

public var preloadString:String=new String();
```

## 定义名为 preloadScreen 实例

这里要定义的 preloadScreen 是 BasicScreen 类的一个实例，本章往后我们深入的时候会用到它，将以下代码添加到 com.efg.framework.GameFrameWork.as 文件的变量定义区。

```
public var preloadScreen:BasicScreen;
```

## 将预加载状态设置为 Main 文档的第一个状态

有些时候，你可能需要程序输出显示出加载状态，此时就有必要持续调用预加载状态了，即便你使用的是其他预加载器也一样（如下一节要提到的麻球广告预加载器）。因为你必须以此确保你的代码可以调用你库里的所有素材。如果 STATE\_SYSTEM\_PRELOAD 是你的游戏程序中唯一的预加载状态（即你没有使用麻球广告预加载器等其他预加载内容），你就必须将它设置为游戏状态的第一个。这可以通过修改在 Main.as 中的 init 函数而实现。同样的，你会在后面的深入内容中看到这样做的效果。现在暂时不需要对 GameFrameWork.as 作任何修改，以下是如何将游戏状态转为预加载状态。

```
switchSystemState(FrameWorkStates.STATE_SYSTEM_PRELOAD);
```

## 往 switchSystemState 函数添加内容

现在需要修改一下 GameFrameWork.as 里的 switchSystemState 函数以实现预加载。必须使之能实现将游戏状态机制设置为 FrameWorkStates.STATE\_SYSTEM\_PRELOAD 状态，以下是要添加进 switchSystemState 函数的代码：

```
//*** add these lines

case FrameWorkStates.STATE_SYSTEM_PRELOAD:

    systemFunction = systemPreload;
```



```
break;
```

```
//*** end add these lines
```

## 添加名为 `systemPreload` 和 `addSounds` 的新函数

`systemPreload` 函数用以确保在游戏运行到时间轴的第三帧之前已加载整个 SWF 文件。当游戏运行到第三帧时，播放指针离开第二帧，并从此开始允许代码层访问素材容器中的所有素材。

```
private function systemPreload():void {
    if (!preloaderStarted) {
        trace("preload started");
        preloadScreen.setDisplayText("Loading: 0%"); //Changed chapter 12
        addChild(preloadScreen)
        preloaderStarted=true;
    }else{
        var percentLoaded:int=(this.loaderInfo.bytesLoaded/_
        this.loaderInfo.bytesTotal)*100;
        trace(percentLoaded);
        preloadString="Loading: "+percentLoaded+"%";
        preloadScreen.setDisplayText(preloadString);
        if (percentLoaded >99) {
            trace(">99");
            this.play();
        }
        if (currentFrame==3) {
            trace("frame == 3")
            addSounds();
            removeChild(preloadScreen);
            nextSystemState = FrameworkStates.STATE_SYSTEM_TITLE;
            switchSystemState(nextSystemState);
```



```
}  
  
}  
  
}
```

该函数首先判断 `preloaderStarted` 的布尔值，如为 `false`，则对 `preloadScreen` 这一实例对象作出相关设置并将之显示出来。如为 `true`，则跳过以上内容并开始监控 SWF 的预加载，监控过程很简单：调用 `percentLoaded` 这一局部变量并用以显示已加载量。如果超过 99% 以上的内容加载完成，就将游戏状态改为 `FrameWorkStates.STATE_SYSTEM_TITLE`，调用 `play` 方法将播放指针从第一帧开始播放。为什么是 99%？因为在某些罕见的情形下，Flash 不会对 100% 加载完成的内容进行注册。以我们的经验，将临界载入量设置为 99% 可以解决这个问题。如果你不喜欢这样做，就把代码更改为 `if (percentLoaded >= 100)` 吧。

`play` 方法一旦被调用，我们就要先等到 SWF 的 `currentFrame` 属性值变为 3，然后才能开始我们的游戏。对 `SoundManager.addSound` 的调用将实例化库里的声音元件。这些功能是在 `addSounds` 函数中实现的。与此同时，我们可以将状态机更改为下一个状态了。此时 `systemPreload` 就会调用 `addSounds` 方法。不过为了能在游戏中添加正确的声音，需要在 `Main.as` 中覆盖这一方法，关于这点的更多细节后面还会提到：

```
public function addSounds():void {  
  
    //stub only needed in IDE preloading  
  
}
```

## 添加麻球广告预加载器及高分排行榜

对 `com.efg.framework.GameFrameWork.as` 进行更改添加麻球加载广告和排行榜，无论对 Flex SDK 还是 Flash IDE 开发的游戏，方法是一样的。你需要在 `www.mochimedia.com` 创建一个账户，在账户里添加一个游戏，并在你为你的游戏放置你自己的广告和排行榜之前建立一个游戏排行榜。游戏和 board IDs 在下一个例子中是真实显示的，它们在广告中显示。他们只我们的测试账户中工作，但是他通过代码赚到的钱会进入那个帐户。所以一定要改变游戏和 board IDs。你已经被警告了！

麻球功能将在我们的状态机制中添加新的状态。如果你已经阅读了部分加载代码并且你用 Flash IDE 开发，你将会看到添加麻球广告和排行榜的状态。如果你用 Flash IDE 开发，而没有看过那部分章节，那些对你这样做来说很重要。

## 导入麻球包

一旦你注册了麻球帐号，你将可以下载最新的 API 代码文件-- `library.zip`。在这个文档中，你可以找到一个名叫 Mochi 的文件夹。把这个文件夹放到你的项目中。如果你在用 Flex SDK，就把它放到 `/src` 文件夹中。用 Flash IDE，它可以和 FLA 在同一目录，另外，你可以把它添加到类文件夹（就像我们在这本书中完成的代码一样）或者其他任何地方，只要你直接引用添加到你的开发工具的路径。



在 com.efg.framework.GameFrameWork.as 和 Main.as 类的类导入部分，添加这一行：

```
import mochi.as3.*;
```

## 改变 Main.as 的对象类型

Main.as 继承 com.efg.framework.GameFrameWork 通过麻球 API 编译，必须是一个特定的类型。它必须是一个 MovieClip，不是 Sprite，并且它必须被声明为动态的。为你的游戏改变 Main.as 里的构造行：

```
dynamic public class Main extends MovieClip {
```

不久我们将演示隧道惊魂游戏。

## 在框架中添加麻球的相关变量

就像我们做的预加载，我们需要在 com.efg.framework. FrameWorkStates.as 添加一些状态为麻球服务做支持。这些是我们要在类里添加的代码（这些要在你用 Flash IDE 添加 STATE\_SYSTEM\_PRELOAD 之前添加）：

```
public static const STATE_SYSTEM_MOCHI_AD:int = 10;

public static const STATE_SYSTEM_MOCHI_HIGHSCORES:int = 12;
```

我们还需要添加一些新的变量在 GameFrameWork.as，来支持麻球服务。mochiGameID 是麻球参考游戏所支持的 ID。mochiBoardId 是麻球参考游戏排行榜支持的 ID。lastScore 是一个变量，它将获取最后一个玩家所得到的游戏分数。我们将用这个变量来发送分数到麻球排行榜。

```
/** added chapter 12 for Mochi Ads and Highscores

public var mochiGameId:String;

public var mochiBoardId:String;

public var lastScore:Number;
```

## 更改 switchSystemState

改变 GameFrameWork.as 里的 switchSystemState 函数，类似于我们之前在 Flash IDE 里预加载的部分。你必须确保一下两部分被添加到 switchSystemState 函数：

```
case FrameWorkStates.STATE_SYSTEM_MOCHI_AD:

    systemFunction = systemMochiAd;

    break;

case FrameWorkStates.STATE_SYSTEM_MOCHI_HIGHSCORES:
```



```
systemFunction = systemMochiHighscores;  
  
break;
```

## 制作特定的麻球广告变化

systemMochiAd 函数必须被添加到 GameFrameWork.as 里。它调用麻球类，并且请求一个动态显示的广告。如果广告加载完成，或者被跳过，或者出现错误，我们会通过解析（例如：解析度：600x400）和调用一个函数（mochiAdComplete）来解决。在最新的麻球文档中介绍了更多的广告选项和新的功能。这是一个麻球广告功能的基本实现：

```
public function systemMochiAd():void {  
    //format resolution as string example: 600x400  
  
    var resolution:String = stage.width + "x" + stage.height;  
  
    MochiAd.showPreGameAd({clip:this, id:mochiGameId, res:resolution,_  
        ad_finished:mochiAdComplete, ad_failed:mochiAdComplete,_  
        ad_skipped:mochiAdComplete});  
  
    switchSystemState(FrameWorkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);  
  
    nextSystemState = FrameWorkStates.STATE_SYSTEM_TITLE;  
}  
  
/** new Function for Mochi ads chapter 12  
  
public function mochiAdComplete():void {  
  
    switchSystemState(nextSystemState);  
  
}
```

一旦 showPreGameAd 函数被调用，状态机制会转移到 FrameWorkStates.STATE\_SYSTEM\_WAIT\_FOR\_CLOSE 状态。一旦广告加载完成或者被跳过，或者有错误发生，mochiAdComplete 函数将被调用。您可以更改这些个别函数，如果您有需要或希望这样做。例如，你可能希望如果被用户选择跳过时，偷偷在广告或者你的网站中加入这些事件。

在你特定的游戏 Main.as 中，你需要添加下面的代码：

```
mochiGameId = "81e8cd4f0999371e";
```

首先，你为你的游戏设置一个特殊的mochiGameId。同样，当你用这个系统建立新的游戏时，麻球会提供这个ID。此外，你还需要在Main.as里设置正确的状态，以便在框架内提供麻球广告支持。下面的代码是原



来Main.as类里面的：

```
switchSystemState(FrameWorkStates.STATE_SYSTEM_TITLE);
```

我们需要改成这样的代码：

```
switchSystemState(FrameWorkStates.STATE_SYSTEM_MOCHI_AD);
```

麻球会在广告播放时加载你的游戏。但是，如果你仍要显示您自己的预加载，您需要像这样重写 systemMochiAd 函数。

注意：我们把区分一下 Flex SDK 和 Flash IDE 版本。Flex 版本将不需要在广告显示之后调用预加载，因为在 Main.as 加载之前，预加载会加载完毕。Flash IDE 版本可能需要在显示广告之后预加载。在任何情况下，一旦广告显示了，我们就可以通过像下面这样重写 systemMochiAd 函数设置要跳转到所需的下一个状态。

```
override public function systemMochiAd():void {  
    super.systemMochiAd();  
  
    /*** flex sdk version  
  
    nextSystemState = FrameWorkStates.STATE_SYSTEM_TITLE;  
  
    /** flash IDE version  
  
    /** nextSystemState = FrameWorkStates.STATE_SYSTEM_PRELOAD;  
  
}
```

这个函数通过调用 super.systemMochiAd 还是和前面的作用相同。最大的变化就是它以 FrameWorkStates.STATE\_SYSTEM\_PRELOAD 更改 systemState，所以这会出现变化。这只在 Flash IDE 中编写预加载时所需要的。

## 制作特定的排行榜变化

在初始化函数 Main.as 中，要添加以下几行，特别是支持特定游戏的麻球服务。我们只在 GameFrameWork 提供了一般的代码，因此它可以用于任何需要麻球服务的游戏。一些游戏不要使用麻球服务，因此，您可以通过使用下面的代码更改 Main.as 手动打开服务（我们尚未在 Main.as 类里创建出这些，因为这只是示例用的）。

```
mochiBoardId = "ffe2de0ae221a7f4";  
  
MochiServices.connect(mochiGameId, this);
```

我们还需要重写 Main.as 里的 systemGameOver 函数来支持麻球排行榜。我们将在其中使用我们在 GameFrameWork 中创建的 lastScore 变量。若要使排行榜运行，我们要在游戏类里创建并设置一个名为 lastScore 的变量，我们可以参考 Main（您稍后将在隧道惊魂中看到这些被使用）。然后我们设置



nextSystemState 为 FrameworkStates.STATE\_SYSTEM\_MOCHI\_HIGHSCORES，这样它的下一步将调用该函数描述。当我们创建了 Main.as 的隧道紧急时，我们会补充下列代码。

```
override public function systemGameOver():void {  
    super.systemGameOver();  
    lastScore = game.lastScore;  
    nextSystemState = FrameworkStates.STATE_SYSTEM_MOCHI_HIGHSCORES;  
}
```

最后，我们需要添加到 systemMochiHighscores 和 mochiHighscores 函数中。这些核心是对 MochiScore.showLeaderBoard 的调用。此调用是麻球提供的标准的函数调用。在麻球服务 API 中有很多其他函数，但最基本的是支持一个排行榜。将这些行添加到 com.efg.framework 包中的 GameFrameWork.as 文件中。

```
public function systemMochiHighscores():void {  
    var resolution:String = stage.width + "x" + stage.height;  
    var o:Object = { n: [15, 15, 14, 2, 13, 14, 0, 10, 14, 2, 2, 1, 10, 7, 15, 4],_  
    f: function (i:Number,s:String):String { if (s.length == 16) return s;_  
    return this.f(i+1,s + this.n[i].toString(16));}};  
    MochiScores.showLeaderboard({boardID: mochiBoardId, score: lastScore,_  
    onClose:mochiHighscoresComplete, res:resolution });  
    switchSystemState(FrameworkStates.STATE_SYSTEM_WAIT_FOR_CLOSE);  
    nextSystemState = FrameworkStates.STATE_SYSTEM_TITLE;  
}  
  
/** new Function for Mochi ads chapter 12  
public function mochiHighscoresComplete():void {  
    switchSystemState(nextSystemState);  
}
```

当用户完成后提交分数或关闭且没有提交而关闭排行榜时，mochiHighscoresComplete 函数将被调用。此函数只是将状态机设置回 FrameworkStates.STATE\_SYSTEM\_TITLE 状态。



## 创建我们自己的 Viral Game

让我们看看一个真实的方案，实际上可能会出现在 viral Flash 游戏产业。我们要假设您作出了一些其他游戏，并创建您自己的框架和可重复使用的类结构，类似于介绍这本书的第 11 章（和这一章的第一节）。

一个客户端电子邮件在星期日晚上约下午 6:00 到达。在电子邮件中是一个简单的游戏引擎的紧急请求和 48 小时的期限。客户需要一个街机风格的游戏与没有射击的游戏，游戏每一关的时间应不超过约 60 秒。你还被告知，控制必须非常容易和简单。您没有其他任何对该游戏的内容描述，但您还被告知游戏必须使用麻球广告和排行榜。客户也没有提供的资源，但希望游戏有一个很基本的 8 位的外观和感觉。我们将创建图 12-8 的游戏画面。

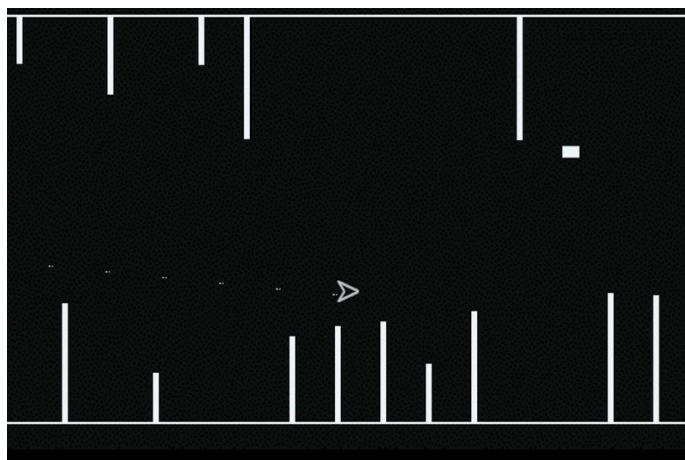


图 12-8. 隧道惊魂游戏截图

## 设计隧道惊魂游戏

这是一个我们的隧道惊魂游戏基本的设计文档：

**游戏名字：**隧道惊魂

**游戏灵感：**Flash 风格经典的游戏就是一个基于技能滚动的街机游戏，玩家使用一个键来控制屏幕上的头像。在这些类型的游戏中，玩家需要跳过障碍，而在其他游戏中，玩家驾驶飞船才能通过洞穴。我们要做一个类似于后者的游戏。其思想是快速结束游戏，张贴高分让全世界的用户看到。

**游戏意图：**尽你的努力通过滚动隧道逃避的障碍飞的更远。障碍的三种类型：从隧道的顶部向下延伸的，从隧道底部向上伸展的，在隧道中间的。

**游戏控制：**空格键将控制玩家的飞船上移。这里没有其他按键。重力将会不断的把飞船拉到屏幕底部。

**游戏过关：**这里没有明确的游戏等级，但每 10 秒，障碍会变的更丰富（更长），颜色的变更，游戏的速度



将会增加。

**游戏结束：**当玩家被障碍碰到，游戏就结束了。

**游戏得分：**玩家会在战斗的每一秒中得到分数。

我们现在来谈一下基本技术设计的信息。

我们现在知道我们要创建的游戏类型，我们有 48 小时去做。在这有限的时间里，我们需要就如何进行技术设计做一些重要决策。首先，让我们讨论一下滚动。我们可以使用一个基于拼贴的滚动引擎（请参阅第 10 章）或我们可以去像第 11 章所载的基于屏幕的引擎。即使这些引擎都已创建，并有必要的类已经是包结构的一部分，我们可能不甚至需要使用它们。

在游戏中移动的障碍是重要的项目，这个游戏世界真的只是一个从右到左移动的滚动的障碍。如果这样做，我们只需使其显示为玩家从左向右移动，当在现实中，玩家在屏幕中间停留移动。我们将通过添加一些排气微粒移动和淡出玩家的背后来为运动做一些额外的提示。

注意：我们将在下一节中展示大量的代码作为一个示范功能。隧道惊悚的整个设置代码在“书写 TunnelPanic.as 类”这一节。

## 创建 PlayerSprite 对象

为我们的玩家，我们将创建一个使用标准的 Flash 动画层实例的 32 \_ 32-pixel 的 Sprite。使用矢量绘图绘制子画面。玩家的船将保持在屏幕的中间，会被重力吸引下来。空格键将用于将对象上移。所需的玩家的变量，如下所示：

```
private var playerSprite:Sprite = new Sprite();  
private var playerShape:Shape = new Shape();  
private var playerSpeed:Number = 4;  
private var playerStarted:Boolean = false;  
private var gravity:Number = 1;
```

playerShape 变量将包含我们为玩家绘制的形状的矢量画布。playerSpeed 表示游戏玩家按下空格键时这艘船将上移的像素数。playerStarted 变量将用于一个新的游戏已经开始前阻止玩家输入。

重力值将被添加到 y 属性的每个帧的刻度中，当玩家按下空格键时，playerSprite 和 playerSpeed 将抵消这重力。



## 创建游戏区域

播放区域将是屏幕组成顶部和底部的水平线的一个非常简单的设计。将会有大约 20 像素缓冲区在顶部和底部的屏幕和游戏区域之间。playfieldSprite 将被添加到显示列表中，并且包含 playfieldShape (白水平线)。

```
private var playfieldSprite:Sprite = new Sprite();
private var playfieldShape:Shape = new Shape();
private var playfieldminX:int = 0;
private var playfieldmaxX:int = 599;
private var playfieldminY:int = 21;
private var playfieldmaxY:int = 378;
```

## 添加障碍

将创建一个池 1\_1-pixel BitmapData 对象。每个将设一个位图对象内。障碍将在屏幕的右侧启动，并向左移动。这将模拟滚动。当一个障碍从非活动状态的池拉出时，它被放在屏幕上之前将被染色和大小调整。

```
private var obstaclePool:Array = [];
private var obstacles:Array = [];
private var tempObstacle:Bitmap;
private var obstaclePoolLength:int = 200;
```

## 玩家飞船的尾气动画

玩家的飞船会从后面排放废气。这只是一种装饰，影响游戏播放不值。这些粒子将使用从第 11 章的 BlitArrayAsset 类创建的排气（和较早前提出的框架包结构）。他们将也从包结构成为 BasicBlitArrayParticle 类的实例。这些粒子放进一个池中，并通过一个不同于其他游戏组件的独立画布进行位图的传输。此画布将采用新的技术（对于这本书）称为“污垢矩形”或者“脏矩形擦除”，这在下一节中会解释。

这些变量创建背景，并为我们排气颗粒创建画布：

```
private var backgroundBitmapData:BitmapData = new BitmapData(580, 400, false, 0x0000000);
private var canvasBitmapData:BitmapData = new BitmapData(580, 400, false, 0x0000000);
private var canvasBitmap:Bitmap = new Bitmap(canvasBitmapData);
```



```
private var blitPoint:Point = new Point(0, 0);
```

The following are used for the exhaust pool (active and inactive):

```
private var exhaustPool:Array = [];
```

```
private var exhaustParticles:Array = [];
```

```
private var tempExhaustParticle:BasicBiltArrayParticle;
```

```
private var exhaustPoolLength:int = 30;
```

```
private var exhaustLength:int;
```

exhaustAnimationList 数组 BitmapData 对象的数组，包含 10 帧的排气颗粒的淡入淡出。

```
private var exhaustAnimationList:Array = [];
```

```
private var lastExhaustTime:int = 0;
```

```
private var exhaustDelay:int = 100+((obstacleSpeedMax * 10) - (10 * obstacleSpeed));
```

图 12-9 是另一种隧道惊魂的游戏画面。此版本现在已注明的游戏要素。

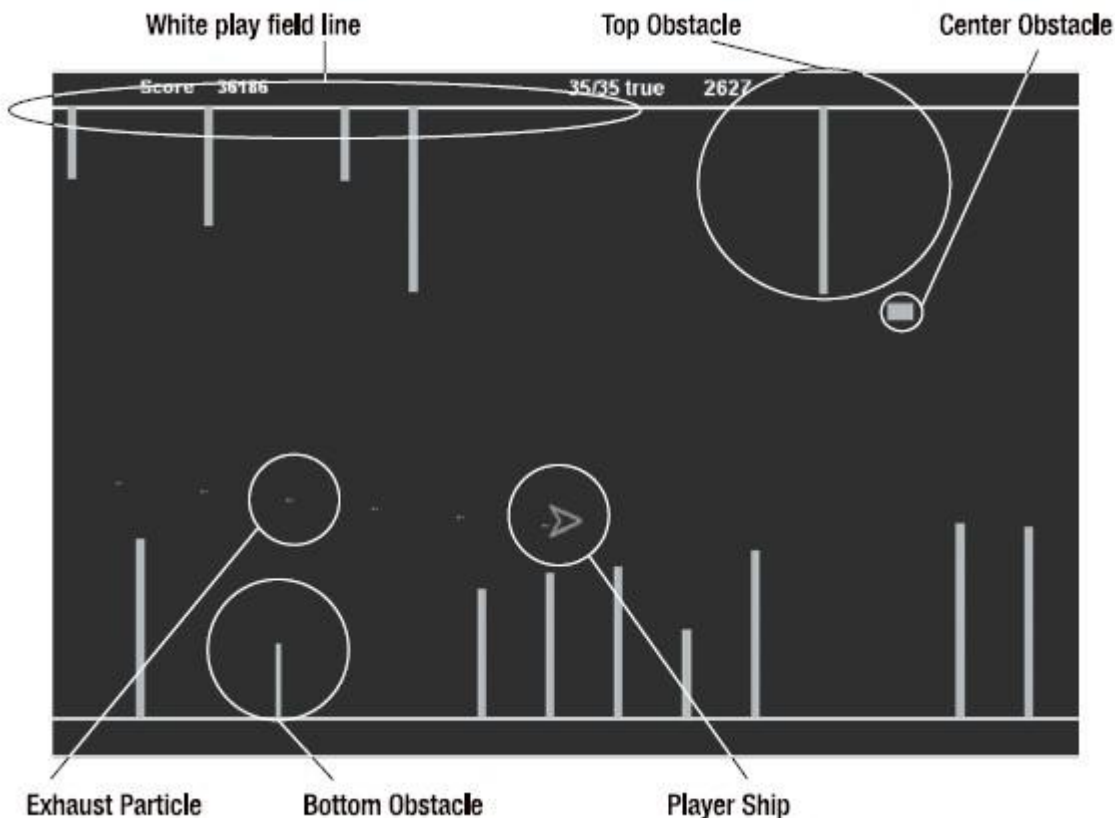


图 12-9.附加说明的隧道惊魂游戏画面



## 使用脏矩形擦除

一种名为“脏矩形橡皮擦”的技术曾用于仅仅更新画布中需要改变的那一部分，而不需要在每一帧之间更新整个画布。这种技术很棒，但它会导致粒子的更新和渲染函数的功能混淆在一起。

我们要做的是在更新函数中删除粒子的原始位置，除了在渲染函数的新位置的 blit 操作。以下是更新函数的代码：

```
exhaustLength = exhaustParticles.length - 1;

canvasBitmapData.lock();

for (ctr = exhaustLength; ctr >= 0; ctr--) {

    tempExhaustParticle = exhaustParticles[ctr];

    //dirty rect blit erase

    blitPoint.x = tempExhaustParticle.x;

    blitPoint.y = tempExhaustParticle.y;

    canvasBitmapData.copyPixels(backgroundBitmapData,tempExhaustParticle.bitmapData.rect,
    blitPoint);

    if (tempExhaustParticle.update(timeBasedModifier)) {

        //return true if particle is to be removed

        tempExhaustParticle.frame = 0;

        exhaustPool.push(tempExhaustParticle);

        exhaustParticles.splice(ctr,1);

    }

}

canvasBitmapData.unlock();
```

您会发现此 blit 是非常类似于前面几章的 blits。我们只需 blitPoint，是当前位置（x 和 y）的粒子。它是复制背景矩形到 canvasBitmapData，然后删除。这有效地清除只有部分需要更新的 canvasBitmapData 而不是整个 canvasBitmapData。



## 增加游戏难度

当游戏运行时，我们每隔 10 秒会增加其难度（obstacleUpgradeWait 的值）。具体而言，我们将更改颜色、频率和障碍的速度。lastObstacleUpgrade 变量将保留从 getTimer 函数调用的返回值最后一次的难度增加。

```
private var obstacleUpgradeWait:int = 10000;
private var lastObstacleUpgrade:int;
```

从池中拉出障碍，并放置在右侧，基于 obstacleDelay 的值播放区域播放时启动。这是要等待障碍之间的毫秒数。每隔 10 的秒随着整个障碍升级此 obstacleDelay 将减少 obstacleDelayDecrease 直到它达到 obstacleDelayMin。

```
private var lastObstacleTime:int;
private var obstacleDelay:int = 800;
private var obstacleDelayMin:int = 50;
private var obstacleDelayDecrease:int = 150;
```

障碍被创建时，它都将会出现在顶部、底部或中间的障碍。centerFrequency 的值是创建一个中心的障碍的百分比几率。centerWidth 和 centerHeight 控制中间的所有障碍的尺寸。

```
private var centerFrequency:int = 15;
private var centerHeight:int = 10;
private var centerWidth:int = 15;
```

如果中心障碍不打算创建，将创建一个的顶部或底部的障碍。障碍的高度是 obstacleHeightMin 和 obstacleHeightMax 之间的随机值。10 秒钟后升级障碍，高度从 obstcaleHeightLimit 增加，直到高度达到 obstcaleHeightIncrease 高度。

```
private var obstacleHeightMin:int = 40;
private var obstacleHeightMax:int = 60;
private var obstacleHeightLimit:int = 120;
private var obstacleHeightIncrease:int = 20;
```

所有障碍将从右到左以相同的速度（obstacleSpeed）创建滚动错觉，播放中看到是从左到右移动。每隔 10 秒速度上升 1，直到它到达 obstacleSpeedMax。

```
private var obstacleSpeed:int = 6;
private var obstacleSpeedMax:int = 12;
```



每隔 10 秒，obstacleColorIndex 将会增加 1。这将在 aObstacleColor 数组中更改池中的颜色值来创建的障碍的颜色。

```
private var obstacleColors:Array = [0xffffffff, 0xff0000, 0x00ff00, 0x0000ff, 0x00ffff,0xffff00,
0xffaaff, 0xaaaff9, 0xcc6600];

private var obstacleColorIndex:int = 0;
```

## 游戏结束

如果玩家碰到障碍对象，游戏结束。我们将为此使用标准 hitTestObject 的碰撞检测是因为障碍是自然生成的。每个障碍是只有一个 1\_1-pixel BitmapData 元素，我们已用代码创建了各种对象。如果我们尝试使用 BitmapData.hitTest，我们会发现，碰撞将不会正确检测到，因为 BitmapData.hitTest 仅使用原始 BitmapData 中的像素，不是，我们创建通过简单地设置他 scaleX 和 scaleY 的 BitmapData 对象的可拉伸的版本。当游戏结束时我们要为 Main.as 设置 lastScore 变量向麻球排行榜提交游戏得分。当 checkForEndGame 函数检测到游戏结束 (gameOver == true)，它将设置 switchSystemState 新的状态为 STATE\_SYSTEM\_PLAYER\_EXPLODE。

playerExplodeComplete 函数实际上是设置 lastScore 变量。

```
public function checkforEndGame():void {
    if (gameOver ) {
        playerStarted = false;
        switchSystemState(STATE_SYSTEM_PLAYER_EXPLODE);
        dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND,
Main.SOUND_EXPLODE,false, 1, 8, 1));
    }
}
```

直到玩家对象消失，playerExplode 函数才会被调用。

```
private function systemPlayerExplode(timeDifference:Number=0):void {
    playerSprite.alpha -=.005;
    if (playerSprite.alpha <= 0) {
        playerExplodeComplete();
    }
}
```



这一函数依次调用 `playerExplodeComplete` 函数，完成淡出。设置 `lastScore` 变量（您将在下一节中看到由 `Tunnel Panic` 游戏继承）。这是同一个变量，`Main` 函数会用它在麻球排行榜（`Mochi leader board`）设置分数。

```
private function playerExplodeComplete():void {  
    dispatchEvent(new Event(GAME_OVER));  
  
    lastScore = score;  
  
    trace("lastScore=" + lastScore);  
  
    disposeAll();  
}
```

## 为隧道惊魂创建 Main.as

`Main` 里的一些改变对游戏来说是必要的，在早期的章节中已经介绍我们在哪里隐藏预加载（`preloading`）和麻球的综合服务。我们需要在 `com.efg.framework.Game.as` 里做一些更改，来支持 `lastScore` 变量。让我们先看看那些更改吧！

## 改变隧道惊魂的 Game.as

若要支持使用一种高评分系统（在本例中为 `Mochi leader board`）我们要为 `com.efg.framework.Game` 做一个小更改。我们需要将 `lastScore` 属性添加到该类的变量定义部分。

```
public var lastScore:Number = 0;
```

## 改变隧道惊魂的 Main.as

我们已经讨论了游戏的 `Main.as` 类、麻球广告（`Mochi ads`）、预加载（`preloaders`）和高分所做的更改。接着，我们遍历所有 `Flex SDK` 和 `Flash IDE` 版本的 `Main.as` 文件所需更改的同时。此类的主要变化是以粗体高亮显示的为预加载（`preloading`）和麻球服务（`Mochi services`）添加所需的代码。请注意！这里还有推动画面和声音代码为隧道惊魂做了一些修改。因为我们已经多次讨论过这些，我们将留给你这些更改，让你自己去发现。

但是，我们应该注意到几个有趣的事情。游戏使用基于 `timer` 的计时器和第 11 章中的 `frame-rate profiler`。请参阅这一章中深入的讨论该代码的部分。此外，`systemWait` 和 `systemLevelIn` 的函数是重写我们第 11 章的创建的，目的是为等级过度以添加一个简单的 `alpha tween` 过渡类。

```
package com.efg.games.tunnelpanic  
{  
  
    import com.efg.framework.FrameCounter;
```



```
import com.efg.framework.FrameRateProfiler;

import flash.text.TextFormat;

import flash.text.TextField;

import flash.text.TextFormatAlign;

import flash.geom.Point;

import flash.events.Event;

import com.efg.framework.FrameWorkStates;

import com.efg.framework.GameFrameWork;

import com.efg.framework.BasicScreen;

import com.efg.framework.ScoreBoard;

import com.efg.framework.SideBySideScoreElement;

import com.efg.framework.SoundManager;

import mochi.as3.*;

dynamic public class Main extends GameFrameWork {

    //custom sccore board elements

    public static const SCORE_BOARD_SCORE:String = "score";

    public static var SOUND_TITLE_MUSIC:String = "titlemusic";

    public static var SOUND_IN_GAME_MUSIC:String = "ingamemusic";

    public static var SOUND_EXPLODE:String = "explode";

    public function Main() {

        if (stage)

            addToStage();

        else

            addEventListener(Event.ADDED_TO_STAGE, addToStage,false,0,true);

    }

}
```



```
override public function addedToStage(e:Event = null):void {  
    if (e != null) {  
        removeEventListener(Event.ADDED_TO_STAGE, addedToStage);  
    }  
    super.addToStage();  
    trace("in tunnel panic added to stage");  
    init();  
}
```

```
override public function init():void {  
    trace("init");  
    game= new TunnelPanic();  
    setApplicationBackGround(600, 400, false, 0x000000);  
    //add score board to the screen as the seconf layer  
    scoreBoard = new ScoreBoard();  
    addChild(scoreBoard);  
    scoreBoardTextFormat = new TextFormat("_sans", "11", "0xffffffff", "true");  
    scoreBoard.createTextElement(SCORE_BOARD_SCORE, new _  
    SideBySideScoreElement(200, 0, 20, "Score", _  
    scoreBoardTextFormat, 25, "0", scoreBoardTextFormat));  
    screenTextFormat = new TextFormat("_sans", "16", "0xffffffff", "false");  
    screenTextFormat.align = flash.text.TextFormatAlign.CENTER;  
    screenButtonFormat = new TextFormat("_sans", "12", "0x000000", "false");  
    titleScreen = new  
BasicScreen(FrameWorkStates.STATE_SYSTEM_TITLE,600,400,false,0x000000 );  
    titleScreen.createOkButton("Play", new Point(250, 250), 100, 20, screenButtonFormat,  
0x000000, 0xff0000,2);  
    titleScreen.createDisplayText("Tunnel Panic", 200, new Point(200, 150),screenTextFormat);
```



```
instructionsScreen = new
BasicScreen(FrameWorkStates.STATE_SYSTEM_INSTRUCTIONS,600,400,false,0x000000);

instructionsScreen.createOkButton("Start", new Point(250, 250),100, 20,screenButtonFormat,
0x000000, 0xff0000,2);

instructionsScreen.createDisplayText("Dodge everything\nCan you go far?.",300,new
Point(150,150),screenTextFormat);

gameOverScreen = new
BasicScreen(FrameWorkStates.STATE_SYSTEM_GAME_OVER,600,400,false,0x0000dd);

gameOverScreen.createOkButton("Game Over", new Point(250, 250), 100,
20,screenButtonFormat, 0x000000, 0xff0000,2);

gameOverScreen.createDisplayText("Submit",100,new Point(250,150),screenTextFormat);

levelInScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_LEVEL_IN,600, 400, true,
0xbfff0000);

levelInText = "GO!";

levelInScreen.createDisplayText(levelInText,100,newPoint(250,150),screenTextFormat);

pausedScreen = new
BasicScreen(FrameWorkStates.STATE_SYSTEM_PAUSE,400,400,false,0xff000000 );

pausedScreen.createOkButton("UNPAUSE", new Point(250, 250), 100, 20,screenButtonFormat,
0x000000, 0xff0000,2);

pausedScreen.createDisplayText("Puased", 100, new Point(250, 150),screenTextFormat);

preloadScreen = new BasicScreen(FrameWorkStates.STATE_SYSTEM_PRELOAD,600, 400, true,
0xff0000ff);

//*** Flex SDK Only. Comment out these lines if using the IDE.

soundManager.addSound(SOUND_IN_GAME_MUSIC, new Library.SoundMusicInGame);
soundManager.addSound(SOUND_TITLE_MUSIC,new Library.SoundMusicTitle);
soundManager.addSound(SOUND_EXPLODE,new Library.SoundExplode);

//preloadScreen not needed for Flex SDK projects

preloadScreen.createDisplayText("Loading...",150,new Point(250,150),screenTextFormat);

//set initial game state

switchSystemState(FrameWorkStates.STATE_SYSTEM_MOCHI_AD);
```



```
//sounds added after pre-load in the addSounds() function

//mochi

mochiGameId = "81e8cd4f0999371e";

mochiBoardId = "ffe2de0ae221a7f4";

MochiServices.connect(mochiGameId, this);

//framerate profiler

frameRate = 40;

frameRateProfiler = new FrameRateProfiler();

frameRateProfiler.profilerRenderObjects = 4000;

frameRateProfiler.profilerRenderLoops = 7;

frameRateProfiler.profilerDisplayOnScreen= true;

frameRateProfiler.profilerXLocation = 0;

frameRateProfiler.profilerYLocation = 0;

addChild(frameRateProfiler);

frameRateProfiler.startProfile(frameRate);

frameRateProfiler.addEventListener(FrameRateProfiler.EVENT_COMPLETE,
frameRateProfileComplete, false, 0, true);

}

override public function addSounds():void {

//flash IDE only (uncomment these lines if using the IDE)

//soundManager.addSound(SOUND_IN_GAME_MUSIC, new SoundMusicInGame);

//soundManager.addSound(SOUND_TITLE_MUSIC,new SoundMusicTitle);

//soundManager.addSound(SOUND_EXPLODE,new SoundExplode);

}

override public function frameRateProfileComplete(e:Event):void {

//advanced timer
```



```
trace("profiledFrameRate=" + frameRateProfiler.profilerFrameRateAverage);
game.setRendering(frameRateProfiler.profilerFrameRateAverage, frameRate);
game.timeBasedUpdateModifier = frameRate;
removeEventListener(FrameRateProfiler.EVENT_COMPLETE, frameRateProfileComplete) ;
removeChild(frameRateProfiler);

//frame counter
frameCounter.x = 400;
frameCounter.y = 0;
frameCounter.profiledRate = frameRateProfiler.profilerFrameRateAverage;
frameCounter.showProfiledRate = true;
addChild(frameCounter);
startTimer(true);
}

override public function systemMochiAd():void {
super.systemMochiAd();
//*** flex sdk version
nextSystemState = FrameWorkStates.STATE_SYSTEM_TITLE;
//flash IDE version
//nextSystemState = FrameWorkStates.STATE_SYSTEM_PRELOAD;
}

override public function systemGameOver():void {
super.systemGameOver();
lastScore = game.lastScore;
nextSystemState = FrameWorkStates.STATE_SYSTEM_MOCHI_HIGHSCORES;
}
```



```
override public function systemGamePlay():void {
    game.runGameTimeBased(paused,timeDifference);
}

override public function systemTitle():void {
    soundManager.playSound(SOUND_TITLE_MUSIC, true,999, 20, 1);
    super.systemTitle();
}

override public function systemNewGame():void {
    trace("new game");
    soundManager.stopSound(SOUND_TITLE_MUSIC,true);
    super.systemNewGame();
}

override public function systemLevelIn():void {
    levelInScreen.alpha = 1;
    super.systemLevelIn();
}

override public function systemWait():void {
    //trace("system Level In");
    if (lastSystemState == FrameWorkStates.STATE_SYSTEM_LEVEL_IN) {
        levelInScreen.alpha -= .01;
        if (levelInScreen.alpha < 0 ) {
            dispatchEvent(new Event(EVENT_WAIT_COMPLETE));
            levelInScreen.alpha = 0;
        }
    }
}
```



```
}  
  
}
```

## 用 Flex SDK 创建 Library.as 类

现在，只是添加我们需要的三种声音。当玩家的船击中的一个障碍时，有两个循环音乐和单个声音效果。如果您使用的 Flash IDE，您不需要此类。

```
package com.efg.games.tunnelpanic  
  
{  
  
    public class Library {  
  
        [Embed(source = '../..../assets/startmusic.mp3')]  
        public static const SoundMusicInGame:Class;  
  
        [Embed(source = '../..../assets/ingamemusic.mp3')]  
        public static const SoundMusicTitle:Class;  
  
        [Embed(source='../..../assets/explode.mp3')]  
        public static const SoundExplode:Class;  
  
    }  
  
}
```

## 添加到 Flash IDE 库

把这些相同的 assets 放到库中，然后预加载 ( preloading ) 时，再把他们放到一个 clip 中。如果您尚未阅读这一节，那么请现在就去阅读这一节。

注意：如果您使用的是 Flash IDE，在 Main.as 只需 addSounds 函数。如果使用的是 IDE，请注释掉 soundManager.addSound 函数调用的 init 函数。

## 书写 TunnelPanic.as 类

这一节展示了 Game.as 类和子类的完整代码。正如你将看到的，我们已经讨论了很多关于代码和算法。这个类包含了部分类的导入，部分变量定义和构造函数：大部分变量我们都已经看完了，剩下的部分只有一个调用了 init 方法的构造函数还有 class,package 后的括号。。

```
package com.efg.games.tunnelpanic  
  
{
```



```
import flash.display.*
import flash.events.*;
import flash.geom.Point;
import flash.geom.Rectangle;
import flash.utils.getTimer;
import com.efg.framework.BasicBlitArrayObject;
import com.efg.framework.BlitArrayAsset;
import com.efg.framework.BasicBlitArrayParticle;
import com.efg.framework.CustomEventLevelScreenUpdate;
import com.efg.framework.CustomEventScoreBoardUpdate;
import com.efg.framework.CustomEventSound;
import com.efg.framework.Game;
public class TunnelPanic extends com.efg.framework.Game
{
    public static const STATE_SYSTEM_GAME_PLAY:int = 0;
    public static const STATE_SYSTEM_PLAYER_EXPLODE:int = 1;
    private var systemFunction:Function;
    private var currentSystemState:int;
    private var nextSystemState:int;
    private var lastSystemState:int;
    private var customerScoreBoardEventScore:CustomEventScoreBoardUpdate = new
CustomEventScoreBoardUpdate(CustomEventScoreBoardUpdate.UPDATE_TEXT,Main.SCORE_BO
ARD_SCORE, "");
    //Tunnel Panic game specific
    private var keyPressList:Array = [];
    private var keyListenersInit:Boolean = false;
    //player ship
    private var playerSprite:Sprite = new Sprite();
```



```
private var playerShape:Shape = new Shape();
private var playerSpeed:Number = 4;
private var playerStarted:Boolean = false;
//playfield
private var playfieldSprite:Sprite = new Sprite();
private var playfieldShape:Shape = new Shape();
private var playfieldminX:int = 0;
private var playfieldmaxX:int = 599;
private var playfieldminY:int = 21;
private var playfieldmaxY:int = 378;
//obstacles
private var obstaclePool:Array = [];
private var obstacles:Array = [];
private var tempObstacle:Bitmap;
private var obstaclePoolLength:int = 200;
//game play
//how long to wait before increasing obstacle difficulty
private var obstacleUpgradeWait:int = 10000;
private var lastObstacleUpgrade:int;
//obstacle frequency
private var lastObstacleTime:int;
private var obstacleDelay:int = 800;
private var obstacleDelayMin:int = 50;
private var obstacleDelayDecrease:int = 150;
//center obstacles
private var centerFrequency:int = 15;
private var centerHeight:int = 10;
```



```
private var centerWidth:int = 15;

//obstacle height

private var obstacleHeightMin:int = 40;

private var obstacleHeightMax:int = 60;

private var obstacleHeightLimit:int = 120;

private var obstacleHeightIncrease:int = 20;

//obstacleSpeed

private var obstacleSpeed:int = 6;

private var obstacleSpeedMax:int = 12;

//obstacleColors

private var obstacleColors:Array = [0xffffffff, 0xff0000, 0x00ff00, 0x0000ff, 0x00ffff,0xffff00,
0xfffaaff, 0xaaaff99, 0xcc6600];

private var obstacleColorIndex:int = 0;

private var gravity:Number = 1;

//exhaust blit canvas

private var backgroundBitmapData:BitmapData = new BitmapData(580, 400, false, 0x0000000);

private var canvasBitmapData:BitmapData = new BitmapData(580, 400, false, 0x0000000);

private var canvasBitmap:Bitmap = new Bitmap(canvasBitmapData);

private var blitPoint:Point = new Point(0, 0);

private var exhaustPool:Array = [];

private var exhaustParticles:Array = [];

private var tempExhaustParticle:BasicBiltArrayParticle;

private var exhaustPoolLength:int = 30;

private var exhaustLength:int;

private var exhaustAnimationList:Array = [];

private var lastExhaustTime:int = 0;

private var exhaustDelay:int = 100+((obstacleSpeedMax * 10) - (10 * obstacleSpeed));

//score
```



```
public var score:int = 0;

private var lastScoreEvent:int = 0;

private var scoreDelay:int = 1000;

private var gameOver:Boolean = false;

public function TunnelPanic() {

    init();

}

// end class

// end package
```

下面的是 init 函数，隧道惊魂 ( Tunnel Panic ) 的 init 函数是用来初始化玩家的飞船，绘制游戏区域，创建障碍和尾气池和尾气例子的创建和初始化。

注意：请确保要添加这些新函数在类内部构造函数后。

```
public function init():void {

    this.focusRect = false;

    createPlayerShip();

    createPlayfield();

    createObstaclePool();

    createExhaustPool();

    setUpCanvas();

    canvasBitmap.y = 20;

    addChild(canvasBitmap);

    addChild(playfieldSprite);

}

private function createPlayerShip():void {

    //draw vector ship and place it into a Sprite instance

    playerShape.graphics.clear();

    playerShape.graphics.lineStyle(2, 0xff00ff);

    playerShape.graphics.moveTo(15, 7);
```



```
playerShape.graphics.lineTo(7, 24);
playerShape.graphics.lineTo(15, 19);
playerShape.graphics.moveTo(16, 19);
playerShape.graphics.lineTo(24, 24);
playerShape.graphics.lineTo(16, 7);
playerShape.x = -16;
playerShape.y = -16;
playerSprite.addChild(playerShape);
}

private function createPlayfield():void {
//draw playfield as two simple lines at top and bottom of screen
playfieldShape.graphics.clear();
playfieldShape.graphics.lineStyle(2, 0xffffffff);
playfieldShape.graphics.moveTo(playfieldminX, playfieldminY);
playfieldShape.graphics.lineTo(playfieldmaxX, playfieldminY);
playfieldShape.graphics.moveTo(playfieldminX, playfieldmaxY);
playfieldShape.graphics.lineTo(playfieldmaxX, playfieldmaxY);
playfieldSprite.addChild(playfieldShape);
}

private function createObstaclePool():void {
for (var ctr:int = 0; ctr < obstaclePoolLength; ctr++) {
var tempBitmapData:BitmapData = _
new BitmapData(1, 1, false, obstacleColors[obstacleColorIndex]);
var tempObstacle:Bitmap = new Bitmap(tempBitmapData)
obstaclePool.push(tempObstacle);
}
}

private function createExhaustPool():void {
```



```
//create look for exhaust  
var tempBD:BitmapData = new BitmapData(32, 32, true, 0x00000000);  
tempBD.setPixel32(30, 15, 0xffff00ff);  
tempBD.setPixel32(28, 15, 0xffff00ff);  
tempBD.setPixel32(27, 15, 0xffff00ff);  
var tempBlitArrayAsset:BlitArrayAsset= new BlitArrayAsset();  
tempBlitArrayAsset.createFadeOutBlitArrayFromBD(tempBD, 20);  
exhaustAnimationList = tempBlitArrayAsset.tileList;  
for (var ctr:int = 0; ctr < exhaustPoolLength; ctr++) {  
var tempExhaustParticle:BasicBiltArrayParticle = _  
new BasicBiltArrayParticle(playfieldminX, playfieldmaxY, _  
playfieldminY, playfieldmaxY);  
exhaustPool.push(tempExhaustParticle);  
}  
}  
  
private function setUpCanvas():void {  
canvasBitmapData.lock();  
  
canvasBitmapData.copyPixels(backgroundBitmapData, backgroundBitmapData.rect,  
blitPoint);  
  
canvasBitmapData.unlock();  
}
```

下面的代码设置了一个新的游戏和级别。newGame 函数将重置所有游戏变量如那些定义了的大小和频率的障碍。newLevel 函数设置屏幕上的玩家、启动时游戏的音乐和重置障碍频率计数器。在此版本的游戏只有一个级别,但如果我们想要添加更多级别,此函数将分离为 newGame 特定函数 和 newLevel 特定函数。

```
override public function newGame():void {  
score = 0;  
obstacleColorIndex = 0;  
lastObstacleTime = 0;
```



```
lastExhaustTime = 0;

lastScoreEvent = 0;

obstacleDelay= 1000;

obstacleHeightMax= 10;

obstacleSpeed= 4;

playerStarted = false;

gameOver = false;

playerSprite.alpha = 1;

switchSystemState(STATE_SYSTEM_GAME_PLAY);

//key listeners

if (!keyListenersInit) {

stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownListener);

stage.addEventListener(KeyboardEvent.KEY_UP, keyUpListener);

keyListenersInit = true;

}

updateScoreBoard();

}

override public function newLevel():void {

stage.focus = this;

dispatchEvent(new CustomEventSound(CustomEventSound.PLAY_SOUND, _

Main.SOUND_IN_GAME_MUSIC, true, 999, 8, 1));

addChild(playerSprite);

playerSprite.x = 300;

playerSprite.y = 200;

playerSprite.rotation = 90;

playerStarted = true;

lastObstacleUpgrade = getTimer();

lastObstacleTime = getTimer();
```



```
}
```

此代码创建游戏循环。游戏循环使用第 11 章中介绍的基于 timer 的步骤计时器。这里有 currentSystemState 的两个值：SYSTEM\_STATE\_GAME\_PLAY 和 SYSTEM\_STATE\_PLAYER\_EXPLODE。

```
override public function runGameTimeBased(paused:Boolean=false,_
timeDifference:Number=1):void {
    if (!paused) {
        systemFunction(timeDifference);
    }
}

public function switchSystemState(stateval:int):void {
    lastSystemState = currentSystemState;
    currentSystemState = stateval;
    switch(stateval) {
        case STATE_SYSTEM_GAME_PLAY:
            systemFunction = systemGamePlay;
            break;
        case STATE_SYSTEM_PLAYER_EXPLODE :
            systemFunction = systemPlayerExplode;
            break;
    }
}

private function systemGamePlay(timeDifference:Number=0):void {
    if (playerStarted) {
        checkInput();
    }
    update(timeDifference);
    checkCollisions();
}
```



```
render();
updateScoreBoard();
checkforEndGame();
}

private function systemPlayerExplode(timeDifference:Number=0):void {
    playerSprite.alpha -=.005;
    if (playerSprite.alpha <= 0) {
        playerExplodeComplete();
    }
}

public function checkforEndGame():void {
    if (gameOver ) {
        playerStarted = false;
        switchSystemState(STATE_SYSTEM_PLAYER_EXPLODE);
        dispatchEvent(new
CustomEventSound(CustomEventSound.PLAY_SOUND,Main.SOUND_EXPLODE, false, 1, 8, 1));
    }
}

private function playerExplodeComplete():void {
    dispatchEvent(new Event(GAME_OVER) );
    lastScore = score;
    trace("lastScore=" + lastScore);
    disposeAll();
}
```

此代码允许用户输入。我们只需要用户在 Tunnel Panic 中输入是空格键。当按键被按下，我们减少 playerSprite（使它在屏幕中向上移动）的 y 值。

```
private function checkInput():void {
    if (keyPressList[32]) {
```



```
playerSprite.y -= playerSpeed;
}
}

private function keyDownListener(e:KeyboardEvent):void {
    keyPressList[e.keyCode] = true;
}

private function keyUpListener(e:KeyboardEvent):void {
    keyPressList[e.keyCode] = false;
}
```

接下来是 update 函数。update 函数会在游戏中升级障碍的属性增加游戏难度。它基于 lastObstacleTime 的值增加了新的障碍。最后，它通过障碍列表和 exhaustParicles 循环，更新其在游戏屏幕上的位置。

```
private function update(timeDifference:Number = 0):void {
    var timeBasedModifier:Number = (timeDifference / 1000)*timeBasedUpdateModifier;
    //score
    if (playerStarted && getTimer() > (lastScoreEvent + scoreDelay)) {
        score += (10 + obstacleSpeed);
    }
    //obstacles additions
    if (getTimer() > lastObstacleUpgrade + obstacleUpgradeWait) {
        lastObstacleUpgrade = getTimer();
        obstacleDelay -= obstacleDelayDecrease;
        if (obstacleDelay < obstacleDelayMin) {
            obstacleDelay = obstacleDelayMin;
        }
        trace("obstacleDelay=" + obstacleDelay);
    }
}
```



```
obstacleHeightMax += obstacleHeightIncrease;
if (obstacleHeightMax > obstacleHeightLimit) {
    obstacleHeightMax = obstacleHeightLimit;
}
trace("obstacleHeightMax=" + obstacleHeightMax);
obstacleSpeed++;
if (obstacleSpeed > obstacleSpeedMax) {
    obstacleSpeed = obstacleSpeedMax;
}
trace("obstacleSpeed=" + obstacleSpeed);
obstacleColorIndex++;
if (obstacleColorIndex == obstacleColors.length) {
    obstacleColorIndex = obstacleColors.length - 1;
}
trace("obstacleColorIndex=" + obstacleColorIndex);
exhaustDelay= 100+((obstacleSpeedMax * 10) - (10 * obstacleSpeed));
}

// add new obstacles
var obstaclePoolCount:int = obstaclePool.length -1;
if (getTimer() > (lastObstacleTime + obstacleDelay) && obstaclePoolCount>0) {
    //trace("creating an obstacle");
    lastObstacleTime = getTimer();
    tempObstacle = obstaclePool.pop();
    tempObstacle.bitmapData.setPixel(0, 0, obstacleColors[obstacleColorIndex]);
    //is it going to be in the center?
    if (int(Math.random() * 100) < centerFrequency) {
        tempObstacle.y = 120 + Math.random()*200;
        tempObstacle.scaleY = centerHeight;
```



```
tempObstacle.scaleX = centerWidth;
}else {
tempObstacle.scaleY = randomNumberFromRange(obstacleHeightMin, obstacleHeightMax);
tempObstacle.scaleX = 5;
(int(Math.random() * 2) == 0)? tempObstacle.y = playfieldminY : tempObstacle.y =_
(playfieldmaxY - tempObstacle.height);
}
tempObstacle.x = playfieldmaxX;
obstacles.push(tempObstacle);
addChild(tempObstacle);
}
//update obstacles
var obstacleCount:int = obstacles.length - 1;
for (var ctr:int=obstacleCount;ctr>=0;ctr--) {
tempObstacle= obstacles[ctr];
tempObstacle.x -= obstacleSpeed*timeBasedModifier;
if (tempObstacle.x < playfieldminX) {
tempObstacle.scaleY = 1;
tempObstacle.scaleX = 1;
obstaclePool.push(tempObstacle);
obstacles.splice(ctr, 1);
removeChild(tempObstacle);
}
}
var exhaustPoolCount:int = exhaustPool.length -1;
if (getTimer() > (lastExhaustTime + exhaustDelay) && exhaustPoolCount > 0 ) {
lastExhaustTime = getTimer();
```



```
tempExhaustParticle = exhaustPool.pop();
tempExhaustParticle.lifeDelayCount=0;
tempExhaustParticle.x=playerSprite.x-30;
tempExhaustParticle.y = playerSprite.y-32;
tempExhaustParticle.nextX=tempExhaustParticle.x;
tempExhaustParticle.nextY=tempExhaustParticle.y;
tempExhaustParticle.speed = obstacleSpeed;
tempExhaustParticle.frame = 0;
tempExhaustParticle.animationList = exhaustAnimationList;
tempExhaustParticle.bitmapData = _
tempExhaustParticle.animationList[tempExhaustParticle.frame];
tempExhaustParticle.dx = -1;
tempExhaustParticle.dy = 0;
tempExhaustParticle.lifeDelay = 3;
exhaustParticles.push(tempExhaustParticle);
}
exhaustLength = exhaustParticles.length - 1;
canvasBitmapData.lock();
for (ctr = exhaustLength; ctr >= 0; ctr--) {
tempExhaustParticle = exhaustParticles[ctr];
//dirty rect blit erase
blitPoint.x = tempExhaustParticle.x;
blitPoint.y = tempExhaustParticle.y;
canvasBitmapData.copyPixels(backgroundBitmapData, _
tempExhaustParticle.bitmapData.rect, blitPoint);
if (tempExhaustParticle.update(timeBasedModifier)) {
tempExhaustParticle.frame = 0;
exhaustPool.push(tempExhaustParticle);
```



```
exhaustParticles.splice(ctr,1);  
  
}  
  
}  
  
canvasBitmapData.unlock();  
  
playerSprite.y += gravity;  
  
}  
  
private function randomNumberFromRange(min:int, max:int):int {  
    return(int(Math.random() * (max - min)) + min);  
}  
  
}
```

通常，在 update 函数后执行来碰撞检测和渲染函数。很简单的 Tunnel Panic 碰撞的例子。我们已实施标准的 hitTestObject 函数。我们这样做是因为构成了我们的障碍的 BitmapData 已调整每个障碍的大小、缩放或旋转对象。BitmapData.hitTest 函数将无法正常工作。

```
private function checkCollisions():void {  
    var playerHit:Boolean = false;  
  
    if (playerSprite.y < playfieldminY+10 || playerSprite.y > playfieldmaxY-10) {  
        trace("hit outside bounds");  
        playerHit = true;  
    }  
  
    for each (tempObstacle in obstacles) {  
        if (playerSprite.hitTestObject(tempObstacle)) {  
            trace("hit obstacle");  
            playerHit = true;  
        }  
    }  
  
    if (playerHit) {  
        gameOver = true;  
    }  
}
```



```
private function render():void {  
    canvasBitmapData.lock();  
    for each (tempExhaustParticle in exhaustParticles) {  
        tempExhaustParticle.render(canvasBitmapData);  
    }  
    canvasBitmapData.unlock();  
}
```

最后，您有这些剩下的这些 function 来清理类似的操作：

```
private function updateScoreBoard():void {  
    customerScoreBoardEventScore.value = score.toString();  
    dispatchEvent(customerScoreBoardEventScore);  
}  
  
private function disposeAll():void {  
    //move all obstacles left in active to pool  
    var obstacleCount:int = obstacles.length - 1;  
    for (var ctr:int = obstacleCount; ctr >= 0; ctr--) {  
        tempObstacle = obstacles[ctr];  
        removeChild(tempObstacle);  
        obstaclePool.push(tempObstacle);  
        obstacles.splice(ctr,1);  
    }  
  
    var exhaustCount:int = exhaustParticles.length - 1;  
    for (ctr = exhaustCount; ctr >= 0; ctr--) {  
        tempExhaustParticle = exhaustParticles[ctr];  
        //dirty rect blit erase  
        blitPoint.x = tempExhaustParticle.x;  
        blitPoint.y = tempExhaustParticle.y;  
        canvasBitmapData.copyPixels(backgroundBitmapData, _
```



```
tempExhaustParticle.bitmapData.rect, blitPoint);  
  
tempExhaustParticle.frame = 0;  
  
exhaustPool.push(tempExhaustParticle);  
  
exhaustParticles.splice(ctr,1);  
  
}  
  
trace("disposed");  
  
}
```

我们在游戏里没有用到 `setRendering` 函数，所以它在前面的代码中只是一个占位符。

## 测试！

当您测试这个游戏时，由于游戏并没有预加载很多的内容，您可能不会看到 `preloader` 的效果。您应该看到的是一个麻球服务广告和屏幕上的游戏标题。该游戏是很简单的：只需按空格键飞起来，和避免撞击到任何东西，包括墙。完成游戏时你应该能够提交你的分数到 Mochi Leader Board，因为我们为这个游戏设置了这些。

## 摘要

你以为就你一个人在做 viral Flash games 吗？错，目前市场竞争可激烈了，不过进入这一行业的门槛是很低的，它的大门对每一个愿意为之努力的人敞开着。

你已经学习了很多以你的 viral Flash games 盈利的方式。也学习了制作一个 viral Flash game（隧道惊魂）的具体步骤，以及为其添加预加载器和麻球网相关服务的代码。接下来怎么发展就看你自己了。你能用本章乃至本书学到的知识，帮助自己做出为之自豪的作品吗？

现在你真的学透了吗？如果你敢这么说，就该回头复习下了。请回到第一章，看看该处阐述做第二个游戏的方法论部分，会不会了有更深的感悟？你感觉到了吗：只有自己动手做游戏，才能精通此道？阐述做第二个游戏的方法论的那一部分的精髓就在于告诉你这个道理。在实际中，在你一个一个游戏做下去的过程中，你会体会到这些含义的。归根结底，本书教给你的最终有两点：

**策划：**做好规划。要做一个优秀的游戏（或者其他的应用程序）离不开好的规划。这里的规划不是指设计流程图或是工程文档，而是一系列可重用、更新和扩展的代码框架，该框架可以让你的项目模型化。本书的游戏框架就是一个很好的例子，我们几乎重复使用着它开发出了本书的全部游戏。不过，该框架不是一成不变的，我们一直在更新它，覆盖它以满足具体游戏的具体需要，直到这一框架越来越成熟。与此同时，我们也用同一个它做出了多个不同的游戏。可见，做好规划，搭好游戏框架结构会使后面创建的每一个项目受益匪浅。

**练习：**我们在本书中做了一些很简单的游戏。第一章的扎气球游戏真的只是对雅达利的街机游戏雪崩和美国动视的 Kaboom 稍加修改。这些都是最早的街机游戏和视频游戏。在过去的 12 章节中，你重复，



实践，重构，并且重复更多，直到你做出一些像是街机，休闲和网页游戏为止。我们仅仅通过做游戏才这样做。如果你想，你可以看遍世界上所有的游戏开发书籍，但是除非你把这些语句用于实践，否则你永远也做不出一个游戏。

你已经读完了这本书，但是我们没有为你做什么。如果你还没有这样做，我们鼓励你去访问这本书的网站：[www.8bitrockert.com](http://www.8bitrockert.com) 或者 [www.friendsofed.com](http://www.friendsofed.com)。在这些网站中，你能得到这本书的源码、在论坛里讨论游戏和开发，也可以在网上看一些新的教程和有关 Flash 游戏和 Flash 游戏世界的内容。

在某种意义上来说，我们把第十一个游戏的博弈论留给你。我们在这本书中给你展示过如何做十种不同的游戏，也给你了工具去做你的第十一个游戏。这个游戏会是什么样呢？我们渴望它的出现。